Animating Multiprocessing Programs in The Smalltalk-80 Environment

Kurt B. Modahl
B.S., University of North Dakota, 1971

A thesis submitted to the faculty
of the Oregon Graduate Center
in partial fulfillment of the
requirements for the degree
Master of Science
in
Computer Science & Engineering

June, 1987

The research described in this draft was successfully defended by Kurt B. Modahl, as part of his requirements for the Master of Science degree in Computer Science & Engineering at Oregon Graduate Center. Kurt died as the result of an automobile accident before the final revisions suggested by his examining committee were incorporated. The degree of Master of Science was awarded posthumously by the trustees of OGC in April 1988.

David Maier
Professor and Acting Chair

Robert G. Babb II
Associate Professor

ABSTRACT

Programming in a multiprocessing environment creates additional complexity issues above those encountered in a uniprocessing model. Animation of the underlying software data structures has been shown to help in management of such issues in uniprocessing environments. Animation tools for multiprocessing environments should also be of assistance to software engineers constructing parallel processing software systems. MPA is an environment that supports creation of animations that support multiprocessing applications in the Smalltalk programming environment.

1. INTRODUCTION

Software systems can become exceedingly complex entities whose meaning cannot easily be grasped by their builders, much less by those unfamiliar with their intricacies. This complexity becomes amplified in multiprocessing systems where multiple processes execute independently of each other and may at times need to exchange information, via messages or by updating values in a shared . Multiprocessing software systems are more difficult to develop, test, debug, and maintain than uniprocessing systems because of the additional complexity resulting from the need for independence, synchronization and sharing of information.

The difficulty of programming multiprocessing systems stems largely from lack of appropriate abstraction. The programmer tends to view a program as consisting of discrete objects that must perform as a whole in coordination to accomplish the requirements of the software. The problem with multiprocessing or parallel processing is that the software abstraction of the user rarely matches what the system's architecture presents. This is particularly true for users new to the concepts of programming systems that involve multiple processes.

One approach to controlling this complexity that has become an increasingly discussed research area involves unifying the fields of

graphics, computer languages and software engineering [Grafton and Ichikawa 85]. Dynamic graphical representations of the inner workings of software can simplify complexity by reducing the software abstraction into readily comprehensible graphical images. Observing graphical images representing data and control structures that animate as the programs execute can give software engineers better insight into the functionality and meaning of a system than attained by looking at snapshots of run-time data values or at just the final output. Such self-animating programs should increase programmer productivity significantly by assisting during the test, debug and maintenance phases of the software development cycle. The concept of a "...*software oscilloscope* which makes the invisible visible" has been suggested as holding great potential for promoting understanding and insight into the complexity of software development [Bocker, et.al. 86].

1.1. Objectives of MPA

The project on which this thesis is based is the construction of a simple novice programming environment for simulating MultiProcessing Animations (**MPA**) that allows: animation of the user's program during execution to reveal it's internal behavior, easy construction of these animations, and user interaction with an active process. The main problem with visualizations or animation of programs until now has been

the tedium of constructing appropriate graphical representations that are used once and then discarded. The process of animating a multiprocessing algorithm, beyond the original coding involved, should be relatively easy for the user, and should involve minimal code modifications. Animation facilities should be appropriate for a variety of algorithms and should be easily extensible and reusable.

MPA is a prototype environment, running under Smalltalk-80, that attempts to provide these tools for animation of programs with multiple processes executing the same or different programs asynchronously, possibly communicating with other processes, and accessing shared data structures. The paradigm of 'views' of Smalltalk objects representing data structures that are dynamically displayed as they are updated to create animations has proven successful [London and Duisberg 85].

Although Pegasus is a uniprocessor architecture design and cannot execute as a true multiprocessor, it does support multitasking which can be viewed as multiprocessing on a single processor. This is similar to the software simulator for Intel's hypercube that runs on the Oregon Graduate Center's VAX 11/780.

1.2. Contributions of MPA

The following discussion concerns some of the features that the MPA environment provides.

1.2.1. Visualization and Animation Toolkit

The classes of MPA provide basic graphical entities (along with their traditional functions) for processes, data structures for simple variables and arrays, and semaphores. Processes display their internal state, while Animating Data Structures (**ADS**) provides for animation of data access. Semaphores graphically indicate when a process has referenced them and whether the reference was successful. Graphical links between a process and an ADS provide a path for symbolic animation of data access.

1.2.2. Graphical Construction of Process Views

A graphical specification of process and ADS views is provided to allow positioning and labeling of view diagrams. This establishes the correspondence between an executing process and appropriate view within MPA. The animation process has been kept as simple as possible by providing an interactive interface that allows the user to describe and position view diagrams. The programmer codes programs as part of the normal Smalltalk development process with the exception that

instances of MPA methods are used where visualization or animation is desired. Since the entirety of the Smalltalk system is open and modifiable by the user, these animating classes and methods are open for any tailoring needed to adjust to a specific program model.

1.2.3. Multi-Level Process Views

Three hierarchical levels of views have been provided: at the top level animation and process viewing is switched off: no animation is visible, at the next level the overall system state of each process can be viewed and at a lower level the program animation of the current active process can be observed as its program containing the ADS executes. The changing state of a process will be reflected by corresponding view updates. These levels are under user control during program execution, making it possible to switch dynamically between the two view levels.

1.2.4. Speed Control

Speed control is essential for program visualization environments because the newer workstations permit rapid display of graphics. Setting the speed (actually the amount of slowdown) provides for viewing the animations and visualizations in a slower mode, thereby increasing comprehension of the graphical displays. In order that a user can view an animation as slowly as desired, a provision for adjusting the speed

and stopping the animation entirely has been implemented.

1.2.5. MPA as a Debugger and Inspector

MPA allows for *stepping* through an execution and provides access to Smalltalk's *debugger* and *inspector* facilities, which allows inspection and modification of code during development and testing.

MPA enables the programmer to observe the execution of processes by visual inspection of process and shared data states. Errors detected by *observing the behavior* of the program can be corrected during execution.

1.2.6. MPA as a Performance Monitor Assistant

Although MPA was not intended to provide performance monitoring capabilities, it indirectly, by means of the visual pathways, allows the programmer to observe performance bottlenecks and abnormalities. This is a result of direct observation of a program's visual behavior.

1.3. Overview of Thesis

The next section provides a brief survey of related work in the field of program visualization. Then a programmer's view of the environment and the programming model is presented. A definition of relevant terms is provided. The design and implementation section is then dis-

cussed. A detailed example of a multiprocessing version of the Sieve of Eratosthenes is presented demonstrating how MPA helps the user create an animation for a program and how the *visualization* can assist in debugging and monitoring performance. The last section provides a summary of the thesis and directions for future research.

2. RELATED WORK

The following is not intended to provide a comprehensive survey of all relevant research in *program visualization* or *animation*. The reader is referred to Grafton and Ichikawa's paper for a more detailed review[Grafton and Ichikawa 85]. Dynamic program visualization has been defined as specifying the program in the usual textual manner, with graphics (in this case animations) used to illustrate some aspect of the program or its run-time execution [Myers 86]. For this thesis animation is defined as the dynamic display of graphical images to give the illusion of motion, whereas visualization utilizes graphical displays without animation.

The **Brown ALgorithm Simulator and Animator (BALSA)** is an integrated software environment designed to animate programs [Brown and Sedgewick 85]. Animation views dynamically change in response to *interesting events* identified in the program's code. Scripts (key-stroke history of a program's execution in BALSA) are provided for that allow replay of desired events. Multiple views of the same data structure and the ability to run multiple algorithms are also supported. Construction of animations are labor intensive and BALSA designers intend to automate the process by providing a standard libray of views.

The **Program Visualization (PV)** environment was developed to support and help visualize large program's structure and function [Brown, et al. 85]. It supports multi-level views, speed control and dynamic visualizations of data structures. Data structure views are changed as data values are updated. There is no smooth animation between the old and new views of the updated data structure. The system does provide for the separation between the code and graphics allowing for independent development of each.

Animus is a prototype system, written in Smalltalk-80, that allows construction of an animation from a library of components [Duisberg 86]. Constraints involving time are used to maintain consistency among the elements of a data structure and their supporting graphical views. Smooth animation views between old and updated states of a data structure, as well as animations involving multiple processes are supported. However, construction of an animation to support some pre-existing code appears to be complex. Animus is, at present, a *guru* system and the ability of a novice Animus user to instrument animations of some existing code is unknown.

The **Programming and Instrumentation Environment for Parallel Processing (PIE)** is, as its name describes an environment specializing in creation and monitoring parallel programs[Segall and

Rudolph 85]. A metalanguage, MP, is used by the user to express and manipulate all parallel processing constructs while the sequential constructs are in an existing programming language. A runtime environments provides the support for MP abstractions. Software sensors are used to instrument a program automatically and can also be directed by the user. These sensors are performance oriented, providing timing statistics, but also provides runtime values of variables. Monitoring is multi-level, but no provision for animation is provided.

The **Interactive Parallel Program Monitor (IPPM)** is a software monitor for Intel's Hypercube multiprocessor architecture that collects user-directed trace event information to perform post-execution analysis and display graphical views of program behavior [Brandis 86]. The user interactively selects elements of the collected events to view. Performance analysis statistics are displayed which allows the user to view bar graphs of CPU utilization, inter-CPU message distribution, and animation of message traffic. The display environment allows for *single stepping*, *fast scrolling*, and *rewinding* the execution trace. IPPM was designed as a performance measurement monitor for algorithms mapped to various cube topologies and control structures. Analysis of the statistics provided is helpful in fine tuning algorithms and topologies for performance.

The **GARDEN** environment is intended to provide a *conceptual programming* environment wherein the programmer's graphical design is executed directly [Reiss 87].

"*Programmers should be able to design, code, debug and maintain their system using their own conceptual models.*"

It incorporates many elements of *graphical programming* with the ability to describe graphical data structures that are displayed as the program executes. Programmers use their own visual concepts to model and develop software. Editors are provided that support drawing pictures of programs and data structures. GARDEN does not support animation and display placement is controlled entirely by the environment, which has created layout problems.

3. DESIGN AND IMPLEMENTATION

MPA was designed as a prototype environment to support visualization of multiple processes and animation of accesses to shared data structures during execution. Graphical objects for processes, semaphores, data access paths, and data structures are supplied as basic elements of the toolkit. These objects (except for datapaths) are also represented in the normal Smalltalk environment. The current implementation was done on a Tektronix 4404/5 (Pegasus) workstation.

The programmer creates an animation by interactively designating locations and labels for those elements of the toolkit that are represented in a Smalltalk program. By referencing MPA methods for creation and access, the designated entities are displayed and animated during the execution of the user's program. The user can control the display level of the animation and its speed. MPA supports *stepping* through an execution and control over process execution order.

The following sections describe the menu functions for run-time support. Each of the MPA classes are presented and briefly described. Finally, the problems of dealing with Smalltalk's **Model-View-Controller** paradigm are discussed.

3.1. Run-time System Menu

Access to the environment is supported by a menu in the **MPAnimator** window. The menu commands are chosen by selecting the appropriate item with the mouse. The commands and a brief description of their function are:

speed

Displays the current amount of slowdown in effect and accepts a new user-input integer. A value of 0 produces no slowdown in the system, increasing values produce increasing amounts.

animate

Selection of this item displays a prompt that asks the user to turn the animation *on* or *off*. When the animation is *off* no visualization or animation is displayed.

step mode

Sets step mode on, and puts control of execution order of processes under user control. During execution, a process's access of shared data structures allows the user to halt and bring up a standard Smalltalk *debugger*.

continuous mode

Sets step mode off, and control of process execution order is determined by the control flow of the user's code.

select process

When in *step* mode, selection of this item allows the user to point to a process icon and select it to be executed by clicking of the mouse button. The designated process executes until it yields or terminates. When a process accesses a shared data structure a prompter is displayed and allows the user to continue or halt.

set level

Displays the current display number level in effect and accepts a new user-input integer. A value of 1 sets the level to display the activity at the process level, no animation of shared data structures is shown. A value of 2 sets the level to display process activity and accesses of shared data structures are animated. Has no effect on current display view.

change level

Toggles the display level between 1 and 2. Results in the new view level to be displayed.

create animation

Allows the user to interactively designate the number, locations and labels for processes and shared data structures. Animation paths are also designated by selecting which process or processes access a shared data. Prompts are displayed for designating the labels for semaphores that restrict access to shared data.

PiStartup

Starts execution of a multiprocessing example for calculation of the value of Pi. Requires that the animation for this program be setup prior to selection.

PrimesStartup

Starts execution of a multiprocessing example for calculation of prime numbers. Requires that the animation for this program be setup prior to selection.

GenericStartup

Starts execution of a user's multiprocessing program. Requires that the animation for this program be setup prior to selection.

3.2. MPA Class Hierarchy

Object-oriented languages, such as Smalltalk, are based on a model of programming in which objects that perform actions communicate by way of messages. This abstraction encourages a more natural design methodology than that provided by procedural languages [Cox 86]. Smalltalk supports inheritance, which allows methods to be shared and modified by different classes of objects. This led to encourage a program design where each of the graphical objects are subclasses of the class MPA, as shown in Figure 3.1. This allowed methods of MPA to be

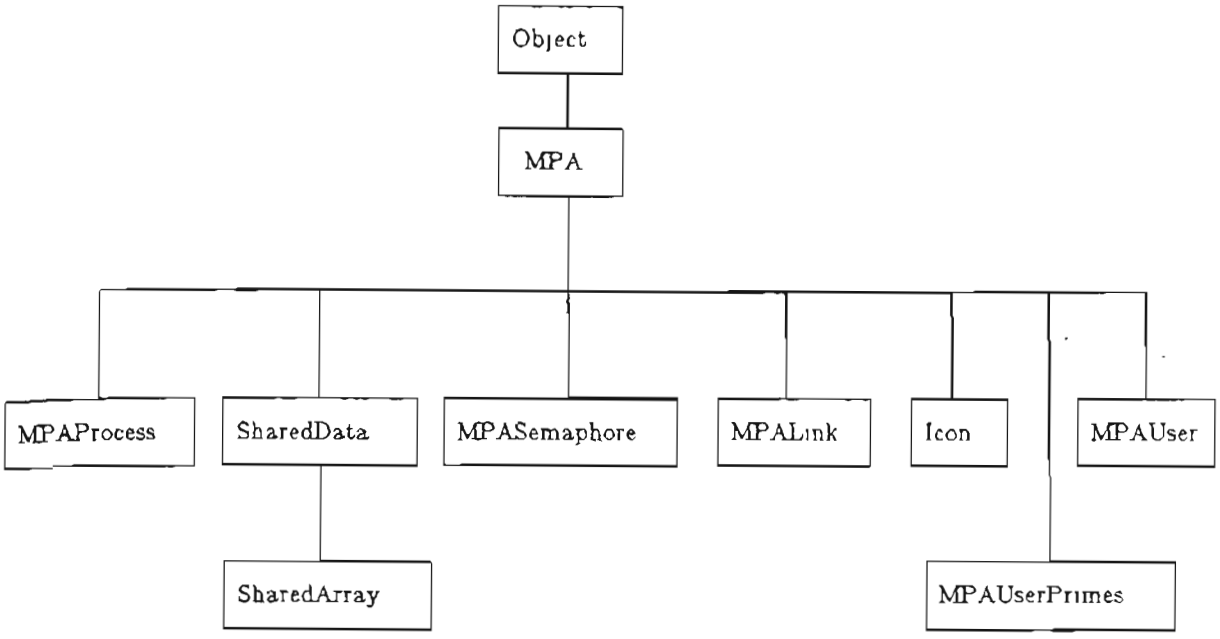

Figure 3.1 MPA Class Hierarchy

used by all of it's subclasses as desired, and for each subclass to override a method if it was required to meet some special need. An explanation of the functionality of each of the classes follows.

3.2.1. Class MPA

The class MPA provides all of the methods for creating an animation:

1) Specification of number, location, label and size for processes and shared data and message view diagrams.
2) Datapaths from processes to shared data and messages.
3) And semaphores used for mutual exclusion.

MPA contains the global class variable data structures for:

1) Icons.
2) Processes.
3) Process, shared data, datapath and semaphore view diagrams.
4) Schedule queue.

Methods for display of view diagrams and level are included and are inherited by all of the MPA subclasses. The methods that are invoked by the run-time menu are also included here.

3.2.2. Class MPAProcess

A process is, "... *a sequence of actions described by expressions and performed by the Smalltalk-80 virtual machine*" [Goldberg and Robson 83]. A process can execute independently of, or synchronize with, other processes. Class MPAProcess supports graphical display of the current state of a process. Each of the possible states of a process (nil, sleep, waiting, active, or terminated) have corresponding icons (class Icon) that are displayed as a process changes from one state to another.

Methods are provided for process creation, changing the state of a process and display of a process's state.

3.2.3. Class SharedData

This class implements the methods that access a shared data structure. Methods are provided that support initialization, addition, retrieval and tests for presence or absence of a value. Execution of these methods invoke various display methods designed to visualize a given operation.

3.2.4. Class SharedArray

This class provides for animation of operations for an arrayed data structure. Graphical display methods are provided that support initialization, retrieval and storage operations. Each element of a shared

array is an instance of class SharedData. This allowed each cell of a large array to be displayed individually in a restricted are of the display screen.

3.2.5. Class MPALink

This class was designed to provide a graphical datapath link between a process and a shared data or message. The state of a link provides visual verification that a process is accessing a shared data or message.

3.2.6. Class MPASemaphore

Semaphores provide for synchronization and mutual exclusion. A process can synchronize with another by issuing a *signal* message to a semaphore, while another process will wait by issuing a *wait* message to the same semaphore. These messages can be given in any order. The process issuing the wait message will suspend itself if the corresponding semaphore has not been given a signal message.

This class provides the graphical support for semaphores. Methods are provided for creation, signal and wait, and display of a semaphore's state.

3.2.7. Class Icon

This class provides for the display of all icons used within MPA. Methods are also provided for editing the bitmaps of icons and for their file storage and initialization. The following icons are used to display the state of a process: Void, Sleep, Wait, Active and Dead. Icons for the state of a semaphore (ISem) and animation of shared data (Datum) are also provided by this class.

3.2.8. Class MPAUser and MPAUserPrimes

These two classes provide examples for the user of MPA. They show the user how MPA methods are used in creating an animation. Class MPAUser provides the methods for an example that uses 5 processes to compute the value of Pi. Class MPAUserPrimes is a multiprocessing version of the Sieve of Eratosthenes that uses 4 processes to compute prime numbers.

3.3. Model-View-Controller Problems

The Model-View-Controller (MVC) abstraction that underlies the Smalltalk programming environment can hinder animation, mostly because of the system overhead involved in multiple MVC schemas and the difficulty in gaining view control when desired to update an animation. Therefore, animation views were implemented in a simpler scheme

(avoiding MVC) so that as much view update control as possible resided within the MPA environment.

MVC facilities were used to create the MPAnimator window and run-time menu as previously described; MPA views were displayed within this window. This had the side effect of actually causing display update problems when continuing from a halt from within MPA. The problem was caused by the updating of the MPAnimator menu window after MPA had updated some of its views. The MPAnimator window re-displayed the previous bitmap saved when the halt occurred, overwriting the new display information. The problem was overcome by reducing the browser menu window so that all MPA views were displayed outside of its boundaries.

Another problem sometimes occurred when the Smalltalk debugger was invoked during an animation. Closure of the debugger window erased parts of underlying MPA display views. This problem was caused when the MPAnimator window was not the current active window, or more properly, when there was no currently active MVC window. Activation of the debugger then resulted in the debugger displaying in the center of the screen. This was solved by establishing the MPAnimator window as the active MVC window by clicking the mouse inside its boundaries.

The avoidance of MVC in updating display views also requires more direct control of all updates. Every update of display information was effected by sending the appropriate display messages to each MPA class entity when it was determined an update was needed. This requires more methods and code than would have been required by using MVC directly, but allowed better control.

4. A PROGRAMMER'S VIEW

MPA provides a simulated multiprocessing environment for prototyping multiple process software systems. The tool allows control over and view of the internal state of processes and shared data structures.

Smalltalk and MPA provide a rapid prototyping environment for multiprocessing. The software model provides for multiple processes that act autonomously and communicate via shared data structures. Mutual exclusion for access to these shared data structures is provided by semaphores. A programmer codes his program using normal Smalltalk methods. MPA provides self-visualizing objects for processes and semaphores, and self-animating objects for shared numbers and arrays. The user utilizes MPA methods and objects wherever *visualization* or *animation* is desired. These normally would be for processes, semaphores, and shared data.

Smalltalk processes running under MPA are in various states during their existence. There is no pre-emption under Smalltalk. A process executes until normal termination, or until it receives a message that causes it to relinquish the CPU. Prior to creation, a process is considered to be in the *nil* state. After creation, a process is in the *sleep* state until it is scheduled for execution by the Smalltalk process scheduler. After a process receives the message *resume*, it is scheduled,

and if no other processes are waiting for execution, it becomes the active process and is allowed to execute by the process scheduler. A process can *yield* to allow other processes to execute, and then enters a *waiting* state, where it is scheduled to execute when the CPU is available, in a first come first served process queue. The last state of a process is *termination*, achieved by natural termination of its last method or by receiving the message *terminate*. [Figure 4.1] here

After a programmer has created his program(s) using MPA methods, the animation display environment must be specified. This is initiated by selecting the *create animation* item from the MPA browser menu [Figure 4.2].

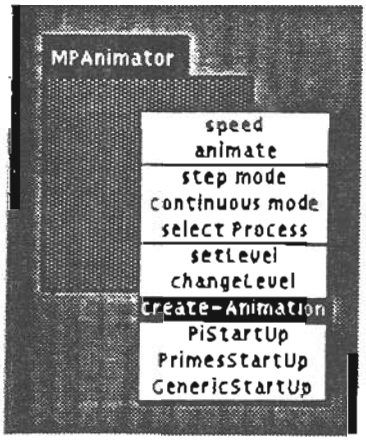

Figure 4.2 MPAnimator menu

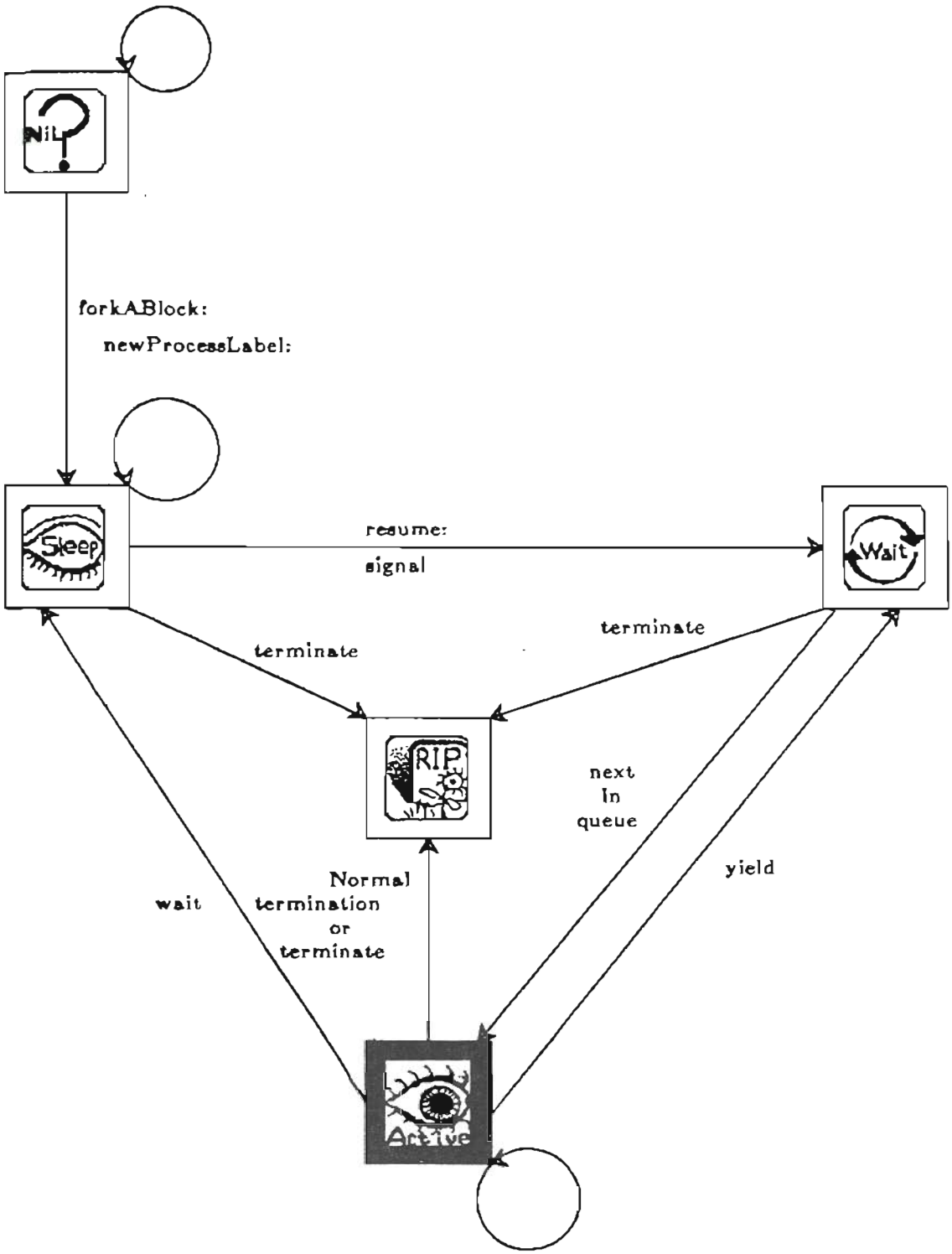

Figure 4.1 State Transition Diagram for MPA Processes.

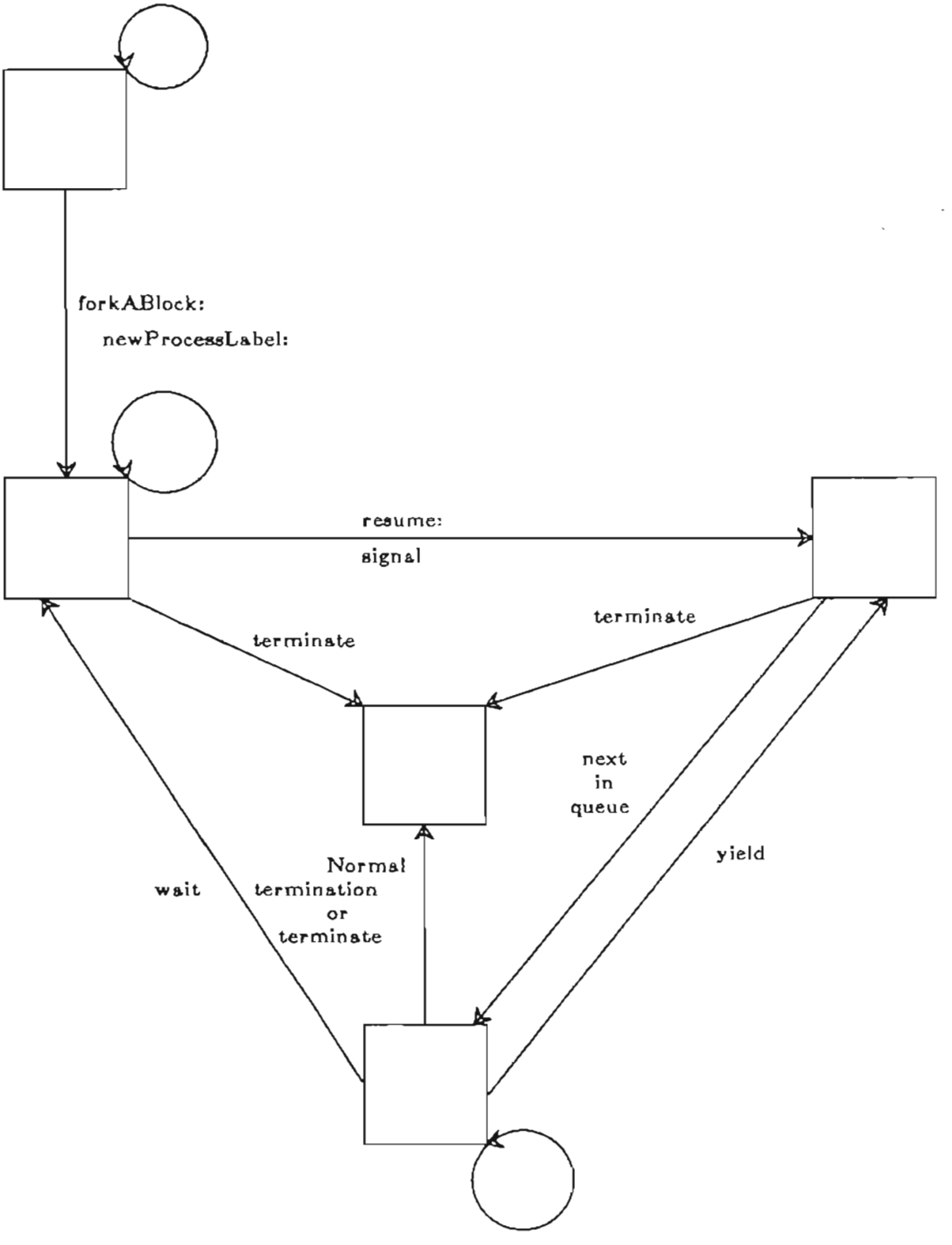

Figure 4.1 State Transition Diagram for MPA Processes.

Selection of this menu item will walk the programmer through animation creation by asking a series of questions. The user is first asked for the number of processes that are to be viewed [Figure 4.3].

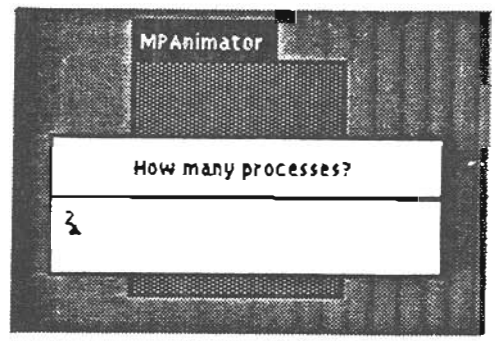

Figure 4.3 Prompter for number of processes.

The programmer is then prompted to select screen locations and labels for each of the processes. Figure 4.4 shows a process location being specified by dragging the crosshair cursor from a starting and ending location.

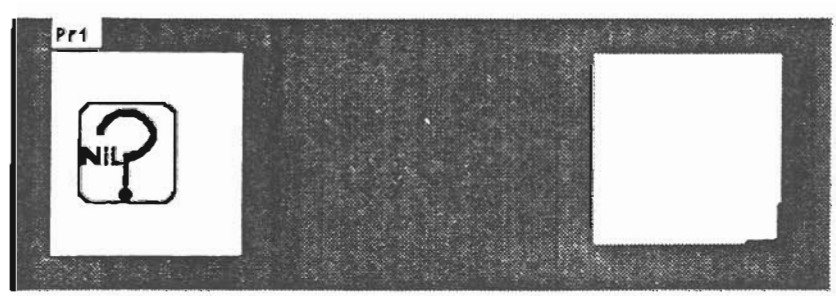

Figure 4.4 Selection of process view location.

Figure 4.5 shows the prompter that asks for the process label after the location and size have been specified.

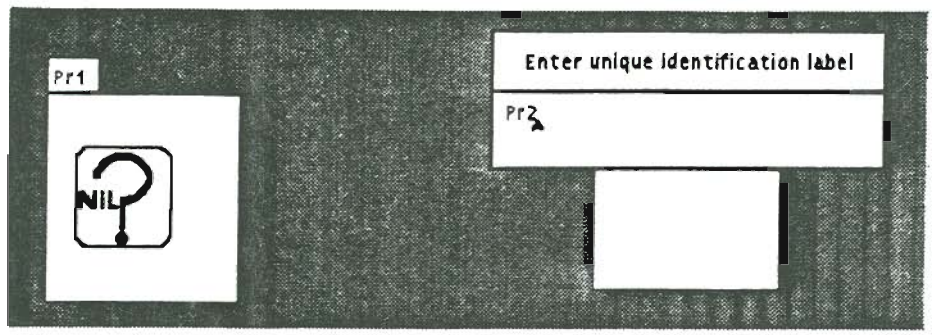

Figure 4.5 Prompter for label.

After the display locations for each of the processes have been specified, the programmer is asked to enter the number of shared data structures to be animated [Figure 4.6]. Shared data structures are represented as graphical entities that have datapath links to the process(es) that reference it. A shared data can have only one link with a process, but can have as many links as there are processes. MPA currently supports animation of shared variables and arrays. For each shared data structure the programmer must supply a location, size, and label in a manner similar to processes. The user also designates datapaths for animation during data structure access by a process, if animation is desired. Figure 4.7 shows the programmer responding to the prompter for displaying datapaths.

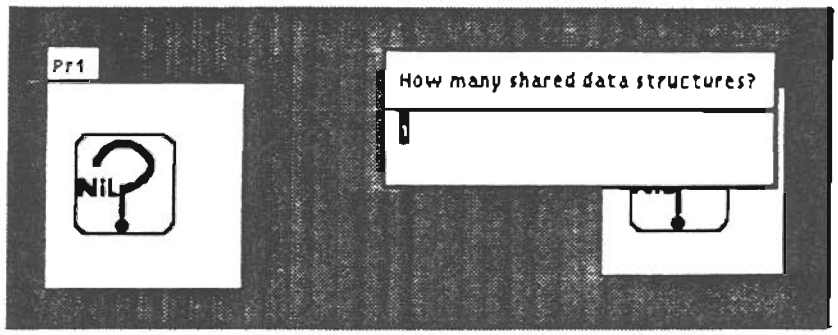

Figure 4.6 Prompter for number of data structures.

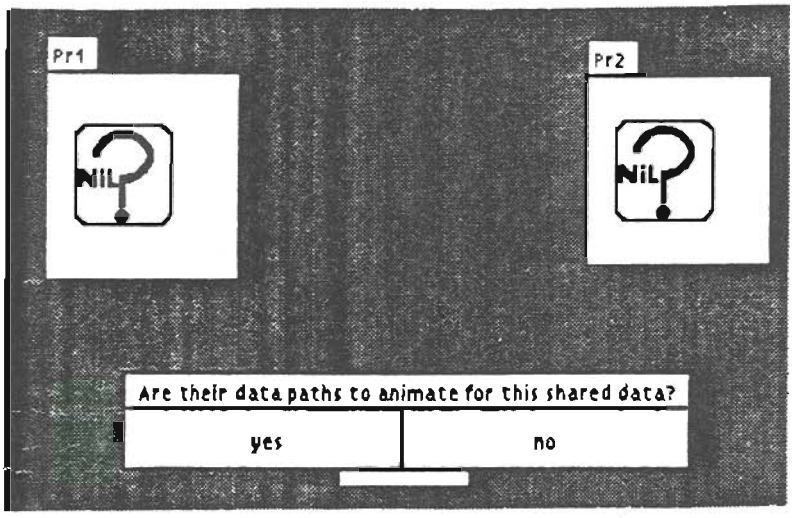

Figure 4.7 Prompter for datapaths.

A datapath link has a starting point in a process and an ending point in the shared data. This is specified by selecting a point (in the display view) for each of the processes that will access the shared data with the mouse. The ending point of the link in the shared data must also be specified. Figure 4.8 shows the MPA programmer about to click the mouse as the crosshair rests on a process view. A process (already specified) is shown with its data link and semaphore displayed. The process of specifying datapaths is terminated by clicking the crosshair on two points outside the views of processes or shared data.

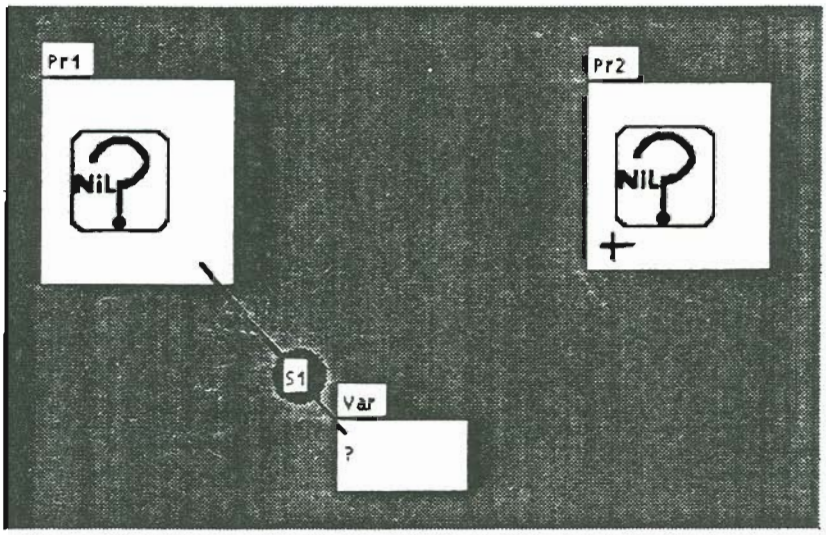

Figure 4.8 Selecting a process for a datapath

If semaphores are to be used for mutual exclusion to the designated data structure, the programmer responds with a *yes* to the prompter as shown in Figure 4.9. This prompter is displayed after each datapath link has been specified. Labels for semaphores are entered by way of a prompter just as they were entered for processes and shared data.

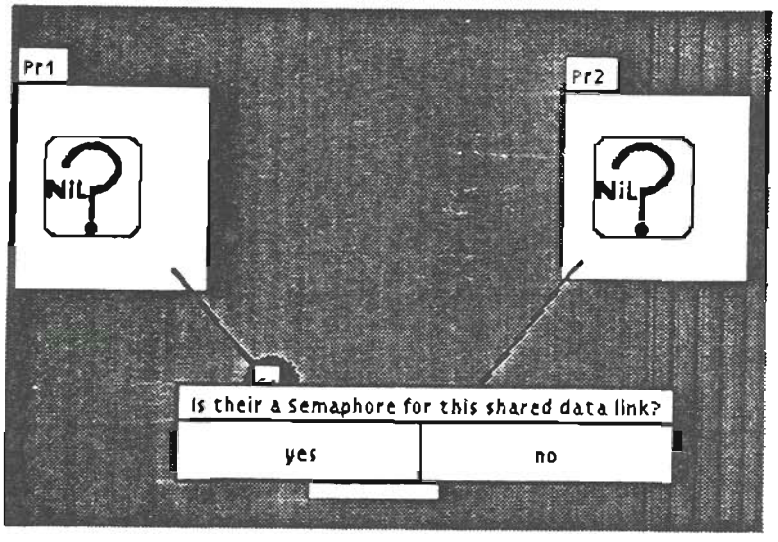

Figure 4.9 Prompter for semaphore.

Shared messages are similar to shared data except that exactly two datapath links must be specified for each shared message. A shared message is a graphical abstraction that contains a single value stored

by one process and referenced by another. The number of shared messages is entered via a prompter as was done for shared data. The location, size, and label are also similarly specified. Displaying datapaths is performed just as it was for shared data, by selecting *yes* in the prompter window. The ending point is specified by selecting a location inside the shared message view. Figure 4.10 shows the second datapath link being specified by clicking the crosshair on the shared message referenced by the process. If semaphores are used to provide mutual exclusion to the shared message, they are specified just as they were for shared data.

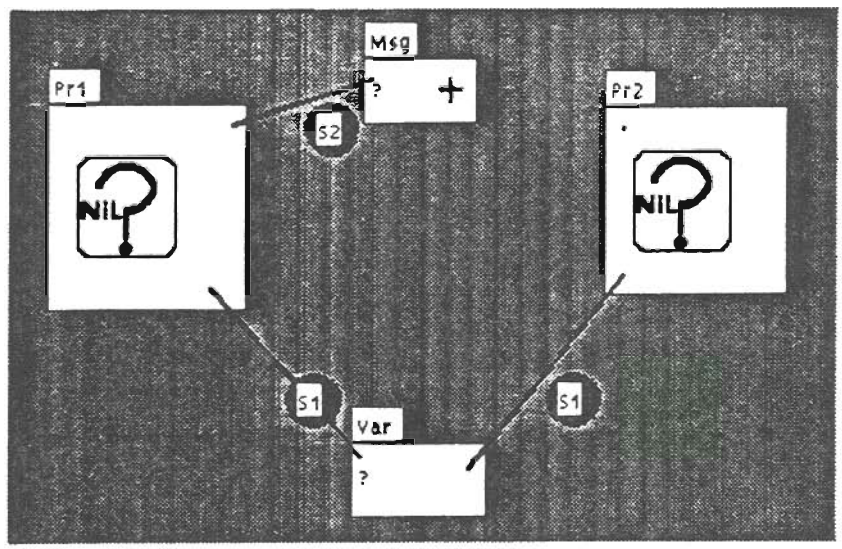

Figure 4.10 Selecting a shared message.

Figure 4.11 shows the finished animation, ready for execution.

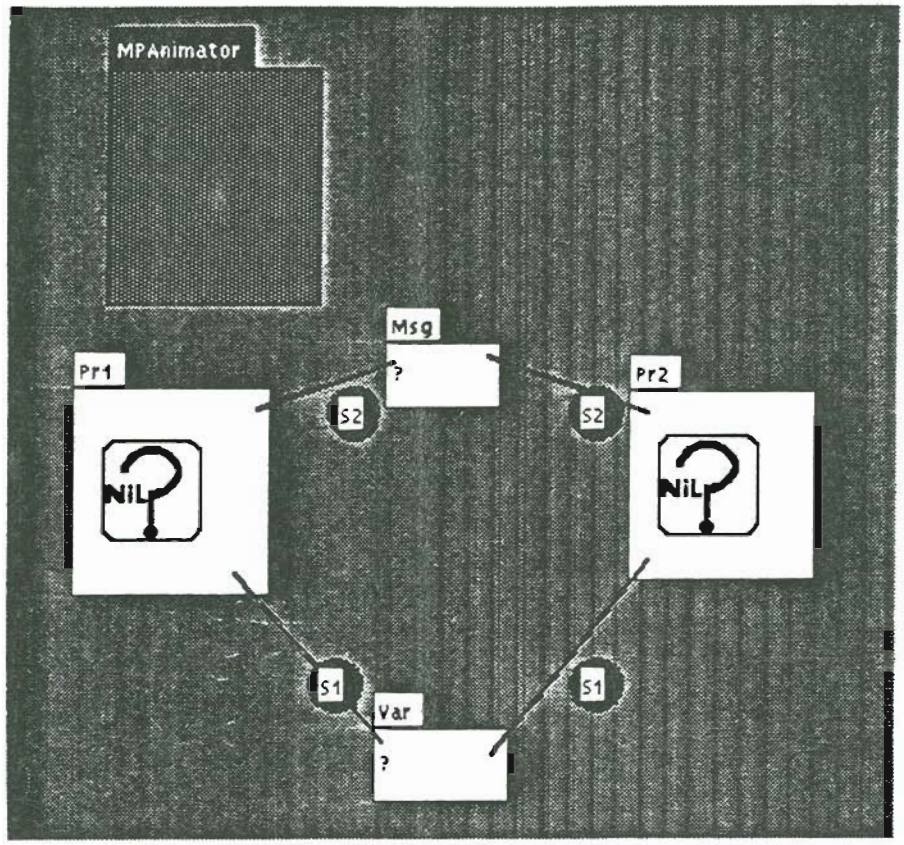

Figure 4.11 Finished animation.

After the animation environment for a program has been described the programmer must set runtime parameters : view level, animation speed, and execution mode.

There are three view levels. The highest level determines whether animation or viewing is shown at all. The first viewing level displays the state of each MPA process. The second viewing level displays the state of each process and animations of the access to shared data as

they are made. The *animate* and *setLevel* menu items are usually set before starting an animation, while *changeLevel* is best used during animation. Figure 4.12 shows the prompter displayed after selecting *animate* in the MPA menu, while Figure 4.13 shows the prompter displayed after selecting *setLevel* in the MPA menu; the current level is indicated and the user enters the new level if a change is desired. Selecting *changeLevel* in the menu also allows the user to change the display level, but works slightly different. It toggles between level 1 and level 2, displaying the levels after the change.

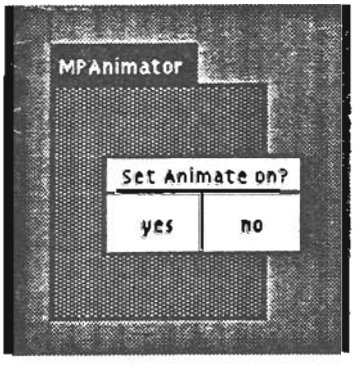

Figure 4.12 Animate prompter.

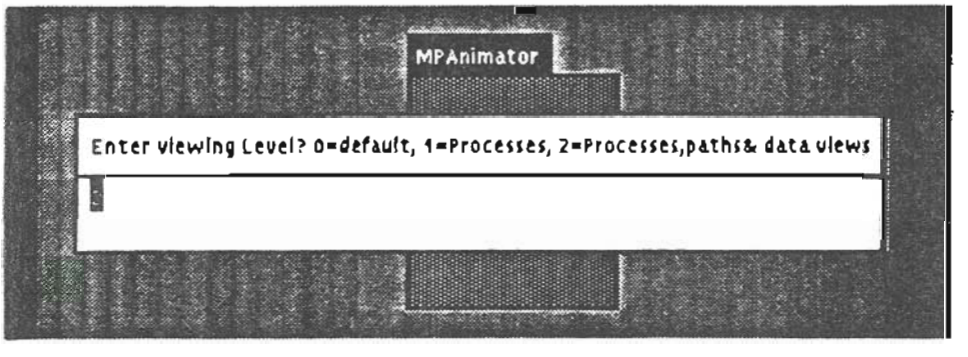

Figure 4.13 SetLevel prompter.

The speed of the animation is determined by the amount of **slow-down** exhibited as MPA updates display views. The current speed is shown in the prompter and a new value is entered by the programmer to control how much slowdown is performed [Figure 4.14]. The value of this feature is of more importance when the programmer is first becoming familiar with the behavior of an MPA application. Later in the development life cycle, increased speed is usually appreciated and can lead to better comprehension.

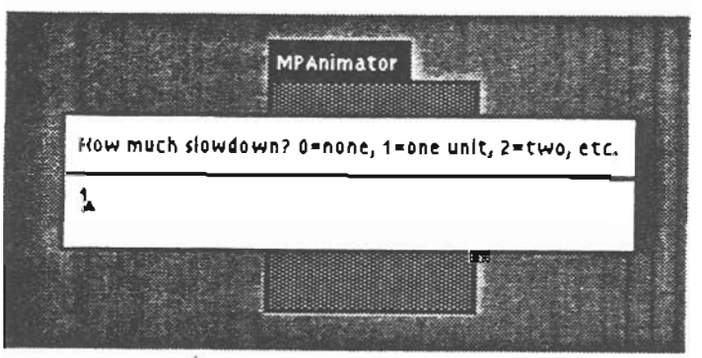

Figure 4.14 Speed prompter.

Prior to executing the program the MPA user must preset the mode of execution to *step* or *continuous*. Step mode requires the programmer to control the order of process execution by selecting a process to execute. Once a process has been chosen for execution via *select process* from the MPA menu, the designated process will execute until it

performs a process *yield* or terminates. A process is designated for execution by positioning the crosshair cursor over a process view and clicking the mouse. Step mode pauses at predetermined points during a process's access of shared data to query the programmer whether to continue or halt [Figure 4.15]. Figure 4.15 shows process P1 pausing prior to updating a shared data. Selecting yes in the prompter causes the process to halt and a Smalltalk debugger window to appear. Continuation proceeds with the execution of the active process until the next access to shared data, whereas halting causes a *halt* to be executed which brings up a standard Smalltalk *debugger*. Selection of *debug* in the debugger window allows examination and modification of the run-time stack and the current methods being executed [Figure 4.16]. The programmer can further examine instance variables in that process's methods.

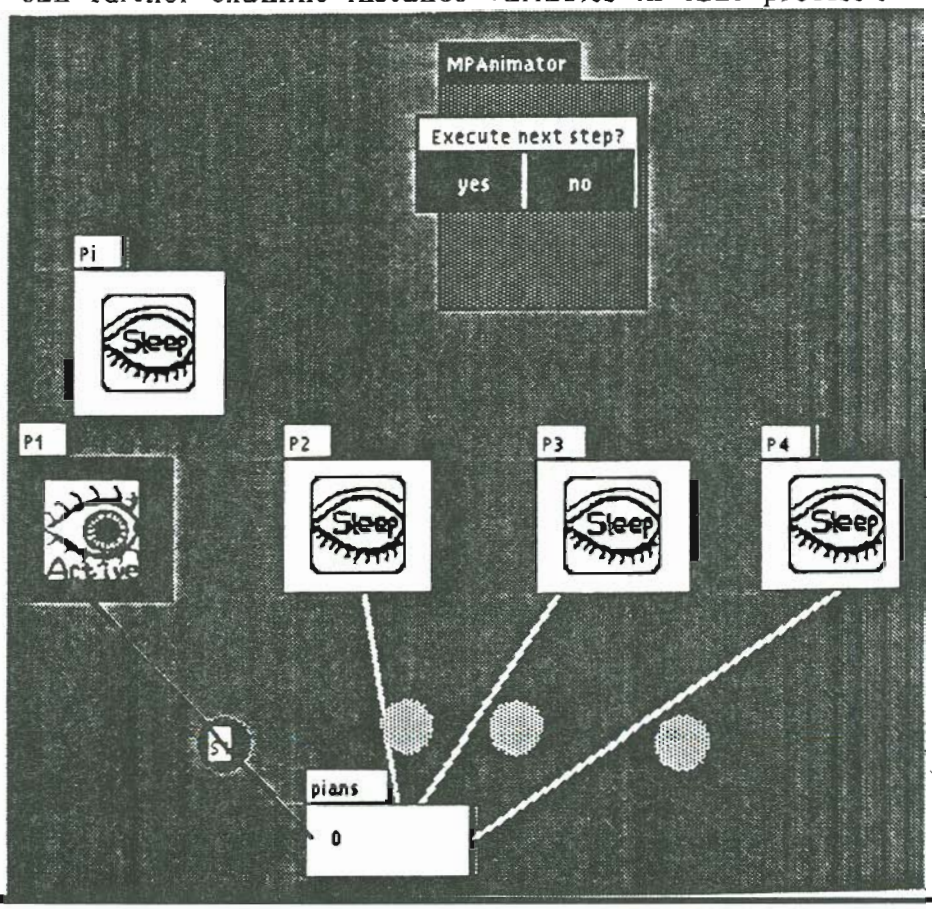

Figure 4,15
step mode prompter for halting execution.

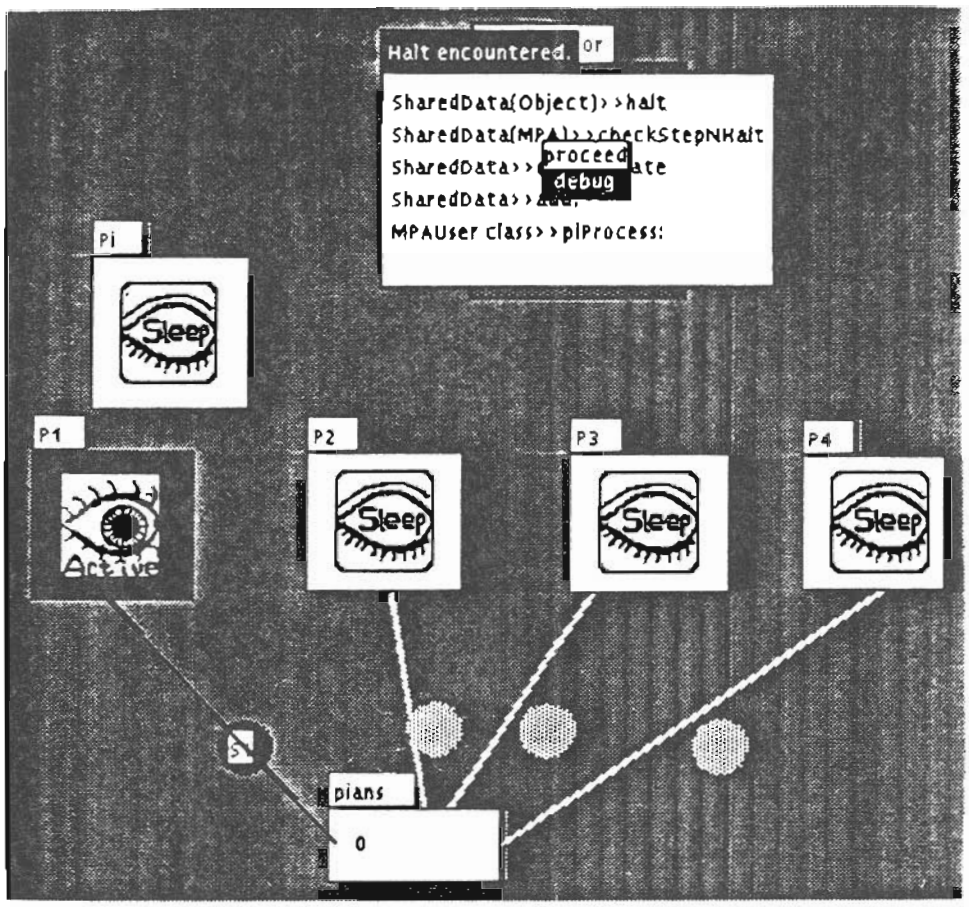

Figure 4.16 Debugger prompter after selecting no in Figure 4.15.

Continuous mode, on the other hand executes the programmer's methods without pauses or queries. Once the primary execution mode has been selected the programmer can start execution of his program by himself or through selection of the *generic startup* menu item in MPA.

As each of the processes is created, scheduled, terminated, or yields, icons for those respective states are displayed at the programmer-designated process location. Similarly, as shared data is initialized, or accessed by individual processes, animated data icons are shown traversing the paths from the relevant process to the shared data. The contents of the shared data is also displayed. If shared data are protected by mutual exclusion then messages by a process that reference semaphores cause the respective semaphore icons to display their state.

5. SIEVE OF ERATOSTHENES

The Sieve of Eratosthenes is a classic algorithm for generating prime numbers. Because of the recent popularity of new parallel processor architectures, the algorithm has been used as a parallelization example several times [Bokhari 87]. Although relatively simple, actual implementation of the algorithm on a modern parallel architecture machine (Sequent Balance 8000) was not trivial [OGC 87]. The algorithm requires extensive synchronization between processes. The bugs and *misbehavior* displayed by the MPA environment are similar to actual parallel programming bugs encountered as mentioned. The *visualizations* and *animations* of MPA illustrate the advantage that graphical environments have for parallel or multiprocessing.

The version that has been used here for demonstrating MPA has what can best be described as a "pipelined" approach. Prime numbers are computed by four sequential communicating processes. The main *Primes* process forks off three new processes. Each of these processes collects a finite number of prime numbers from another process (up to the square of its starting prime) and then communicates potential prime numbers to the next process. The third process simply collects prime numbers that the second process has sent it. After the first process receives and computes the the number of primes it is to find, it

generates possible primes and passes them onto the second process. The second process collects primes from process one up to the square of its starting prime and then passes all subsequent primes it receives onto the third process. Process three collects primes up to the square of its starting prime and then signals *Primes* that it is finished. The main process displays the primes it has found and terminates the other processes. The starting number of primes received by process one determines the number of primes found. For n = 2 , 15 primes are found ; for n = 3 , 99 primes are found.

Each of the computing processes stores the prime numbers it finds in a global array. No mutual exclusion is provided for accessing the shared array since each process has its own, non-overlapping, starting and ending indices. Processes 2 and 3 receive their starting indices, for the global primes array and possible primes, via shared integer values. Access to these shared data are controlled via semaphores. Each process receives its data from the preceding process in a pipelined fashion.

5.1. Creating the Animation

The programmer constructs the animation by specifying view diagrams for the processes, shared data and messages, and semaphores. For shared data and messages, datapaths are also specified. For this

example, there are 4 processes, 1 shared data (an array), and 5 shared messages. Access to each of the shared messages is controlled by semaphores; the shared array has no mutual exclusion protection.

Figure 5.1 shows the completed animation as specified by the programmer, prior to executing the program. Index1 and Index2 are the starting array indices that PR2 and PR3 receive from PR1 and PR2 respectively. Prime1 contains the possible primes that PR1 sends to PR2; Prime2 contains the possible primes that PR2 sends to PR3. The primes array contains the actual primes that are found by the processes after they have been sieved. Done is a value that is set by PR3 and tested by the Primes process to determine when the other processes are done.

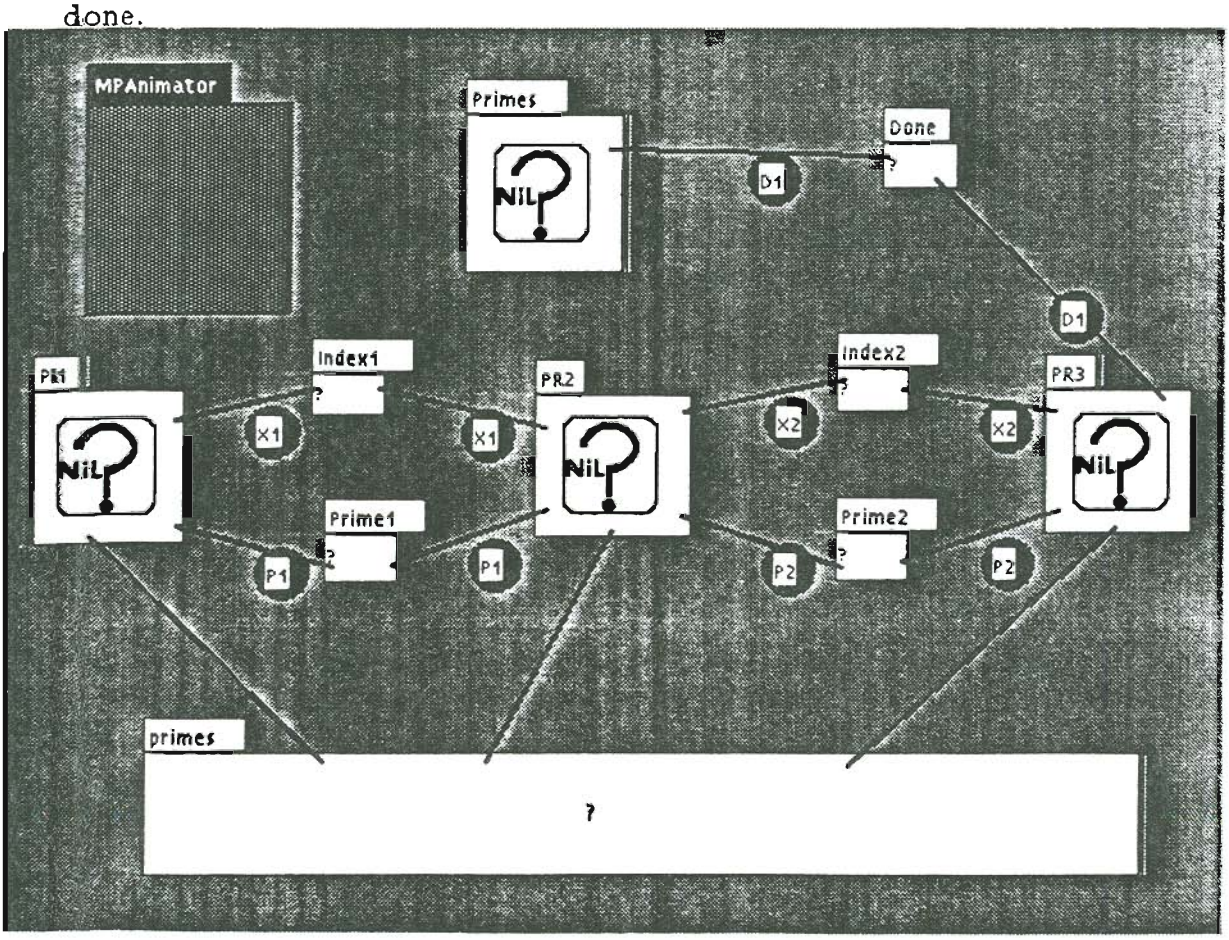

Figure 5.1 Finished animation for Sieve application

5.2. Using MPA Objects

The animation described in Figure 5.1 can be viewed as a design specification diagram that must be coded into Smalltalk methods. The algorithm as presented here requires the use of MPA objects and methods where visualization and animation is desired. In the following sections the algorithm's Smalltalk code containing MPA methods (shown in boldface) will be presented and briefly discussed.

An instance of class MPAProcess is required for each process that is to be visualized during an animation. Since the animation requires visualization of the main Primes process, a startup method is required as follows:

```
PrimesStartUp

"adds startup Primes as a newprocess;
   Primes in turn creates other primesprocesses."
  | primes |
       "forkABlock will display the current level"
  primes <- MPAProcessor forkABlock: [MPAUserPrimes Primes]
                newProcessLabel: 'Primes'.
  MPAProcessor resume: primes.
```

A suspended (state of sleeping) process is created by sending the message forkABlock: [MPAUserPrimes Primes] newProcessLabel: 'Primes' to MPAProcessor. MPAProcessor (instance of class MPA) is a global Smalltalk variable. Execution of this message causes a new process, consisting of the method Primes in class MPAUserPrimes, to be created

and assigned to the local variable primes. At the same time, an internal binding of this process to the process view diagram labeled 'Primes' is also performed by MPA. This establishes the mapping between the programmer's code and the animations that are specified within MPA. MPA will display the Sleep Icon at the view diagram location for 'Primes' indicating that a suspended process has been created. The process (containing the method Primes) is scheduled for execution by the Smalltalk process scheduler by sending the message resume: primes to MPAProcessor. This message causes the display of the Wait icon at the process's view diagram, and immediate execution if there are no other processes active. The Active icon is displayed at a process's view diagram when there are no processes executing and it is the next process the Smalltalk scheduler will execute.

```
Primes
  "This is the main Primes process. It invokes other child processes."
  | x y pr1 pr2  pr3 pr4 wans xans
    yans zans pview numprimes test forever |
      " init  all shared data and messages & semaphores "
  self initSharedData .
  self initSemaphores .
  FillInTheBlank request: 'How many Primes should first process create? '
            displayAt: Sensor cursorPoint
            centered: true
            action: [:num | num <- Number readFrom: (ReadStream on: num).]
             initialAnswer: '2' .
     "get number from string"
  numprimes <- num.
  Debug transShow: 'Primes start seed ****' and: numprimes .
```

Initialization of shared data and messages and semaphores is performed by the methods initSharedData and initSemaphores. A Smalltalk prompter is displayed asking the user for the number of primes the first process should find. This number determines how many primes each of the processes will find. The message transShow: and: issued to Debug is an MPA method that displays its information in the Smalltalk System Transcript window. It is useful for supplying debugging information that is not part of the animation. Figure 5.2 shows the state of the animation after the prompter has been displayed. Primes is the active process and PR1, PR2 and PR3 have not been created yet (state is nil). The shared data and messages have been initialized to 0. The semaphores for the shared messages indicate that they have received a signal (indicated by gray ISem icon), However, the semaphore for the message Done has not received a signal, indicated by the white Isem icon.

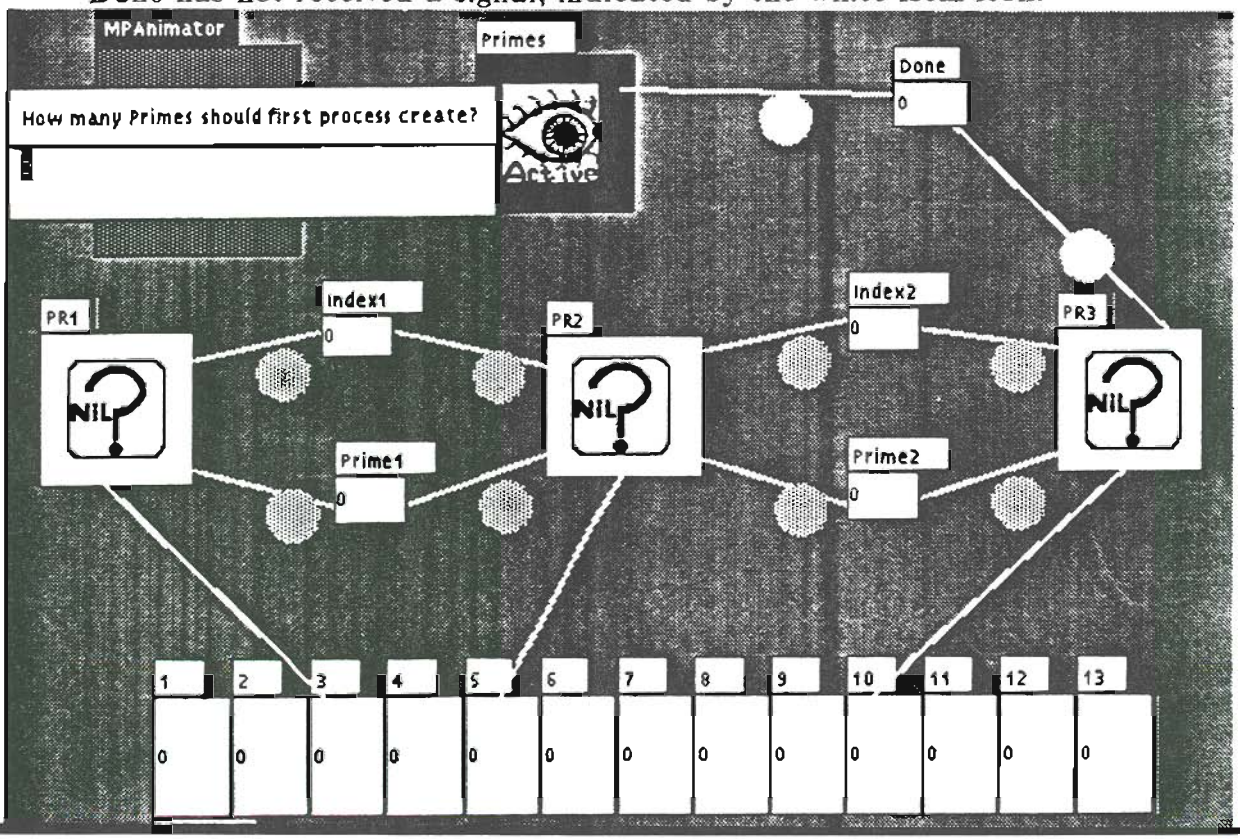

Figure 5.2 Prompter for number of primes

```
pr1 <- MPAProcessor forkABlock: [self primesProcess1: numprimes ]
            newProcessLabel: 'PR1'.
pr2 <- MPAProcessor forkABlock: [ self primesProcess2 ]
            newProcessLabel: 'PR2'.
pr3 <- MPAProcessor forkABlock: [ self primesProcess3 ]
            newProcessLabel: 'PR3'.
    " resume: schedules the processes for execution on the cpu "
MPAProcessor resume: pr1 .
MPAProcessor resume: pr2 .
MPAProcessor resume: pr3 .
```

After the shared data and semaphores have been initialized the Primes process then creates 3 new processes. These 3 processes (pr1, pr2 and pr3) have view diagram labels ('PR1', 'PR2' and 'PR3') that correspond to those specified during animation layout. The appropriate icons for process creation and scheduling is displayed as the methods forkABlock: newProcessLabel: and resume: are executed. The three processes execute three different methods: primesProcess1: numprimes, primesProcess2 and primesProcess3.

```
"Main yields to other PR1-PR3 processes
      that are waiting for the processor"
MPAProcessor yield.
  "WAIT FOR SIGNAL FROM PR3 THAT WE'RE DONE"
forever <- true .
[ forever ]
  whileTrue: [
    " CRITICAL SECTION"
    DoneSem wait .
        ( Done testFor: 0 )
          ifTrue: [  DoneSem signal .
                     MPAProcessor yield . ]
           ifFalse: [ Done set: 0 .
                      forever <- false .
```

```
                    DoneSem signal .
                ]."if testFor"
        " END CRITICAL SECTION"
    ]." while forever"
```

After execution of the resume: messages to the processes, Primes suspends itself from execution by sending the message yield to the MPAProcessor. Primes then enters a state of waiting (indicated by display of the Wait icon) and allows the next process in the queue to execute. When Primes is next scheduled to execute it enters a loop that checks the contents of the shared message Done for a non-zero value (which indicates that the processes are done with their work). Done is a shared message that requires mutual exclusion to control access; this is provided by the semaphore DoneSem. Primes is suspended (a state of sleep) after it sends the message wait to DoneSem if a corresponding signal message has not been sent to the semaphore [Figure 5.5]. If a signal had previously been issued to DoneSem then Primes acquires the semaphore and continues execution. Successful acquisition of the semaphore DoneSem (process returns from execution of the wait method) is indicated on the screen by the display of a black form of the ISem icon and the label for the semaphore, in this case 'D1'. PR3 is responsible for issuing the signal message to DoneSem when it has finished collecting primes. If the contents of Done is zero, it sends a signal message to

DoneSem and suspends itself again. If the value is non-zero, it sets Done to 0, signals DoneSem and exits the loop.

"TERMINATE OTHER PROCESSES STILL SCHEDULED OR WAITING"
MPAProcessor terminate: pr1 .
MPAProcessor terminate: pr2 .
MPAProcessor terminate: pr3 .

The final step for Primes is to terminate the other three processes by sending the message terminate: to MPAProcessor. Execution of this message causes termination of that process and the Terminate icon to be displayed at each process's view diagram. Figure 5.3 shows Primes as the active process and the display after PR1 has been terminated.

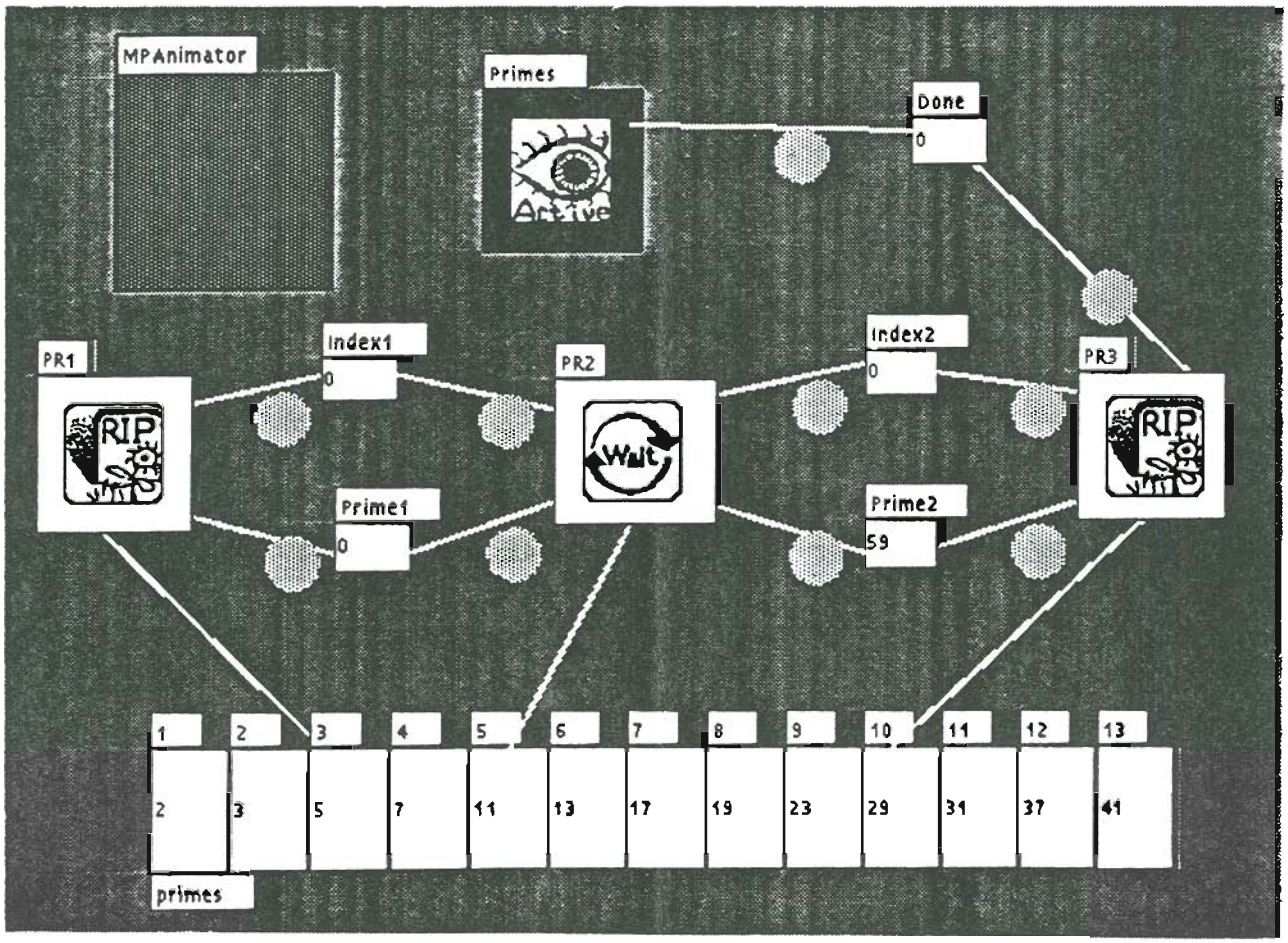

Figure 5.3 Primes terminating other processes.

The Primes process initializes the shared data and messages by executing the method initSharedData.

```
initSharedData

    "Performs shared data initialization of classvars."
    PrimeArray <- SharedArray newArray: 'primes'
                          size: 100 value: 0.
    Index1 <- SharedData newNumber: 'Index1' value: 0.
    Index2 <- SharedData newNumber: 'Index2' value: 0.
    Prime1 <- SharedData newNumber: 'Prime1' value: 0.
    Prime2 <- SharedData newNumber: 'Prime2' value: 0.
    Done <- SharedData newNumber: 'Done' value: 0.
```

PrimeArray is a global class variable that is initialized by sending the message newArray: 'primes' size: 100 value: 0 to the class SharedArray. The contents of the array labeled 'primes' are initialized to 0 and displayed at the appropriate view diagram. Figure 5.2 shows the array and messages after they have been initialized. MPA displays that portion of the array that fits within the view diagram. Index1 is also a global class variable and is initialized by sending the message newNumber: 'Index1' value: 0. The view diagram for 'Index1' is displayed as it is initialized. The other class variables, Index2, Prime1, Prime2 and Done, are initialized in a similar manner.

The Primes process initializes the semaphores to be used by executing the method initSemaphores.

```
initSemaphores
```

```
"Initializes all classvar semaphores used"
"DoneSem does not get an initial signal ."
"label sem & signals set to 0"
DoneSem <- MPASemaphore newSemaphore: 'D1'.
    "label sem & signals set to 0"
Index1Sem <- MPASemaphore newSemaphore: '4.'.
Index1Sem signal .  "set initial signal for other processes"
    "label sem & signals set to 0"
Index2Sem <- MPASemaphore newSemaphore: '4.'.
Index2Sem signal .      "set initial signal for other processes"
    "label sem & signals set to 0"
Prime1Sem <- MPASemaphore newSemaphore: 'P1'.
Prime1Sem signal .      "set initial signal for other processes"
    "label sem & signals set to 0"
Prime2Sem <- MPASemaphore newSemaphore: 'P2'.
Prime2Sem signal .      "set initial signal for other processes"
```

The semaphore DoneSem is declared by sending the message newSemaphore: 'D1' to the MPA class MPASemaphore. The ISem icon is displayed as a white form at the location(s) for 'D1'. Index1Sem is declared and sent a signal message, that causes the ISem icon to be displayed as a gray form (indicating a signal) at the location(s) for '4.'. The other semaphore class variables, Index2Sem, Prime1Sem, Prime2Sem, are declared and signaled in a similar manner [Figure 5.2].

```
primesProcess1: numPrimes

    |startx endx possibleprime forever primefull |
"work  for process  P1 "
"set first few primes in classvar PrimeArray ; not semaphore restricted"
startx <- 1 .
numPrimes == 1
    ifTrue: [  PrimeArray  at: 1 put: 2 .
               endx <-  2 . ]
```

```
ifFalse: [ numPrimes == 2
        ifTrue: [ PrimeArray  at: 1 put: 2 .
                  PrimeArray  at: 2 put: 3 .
                   endx <-  3 . ]
        ifFalse: [ numPrimes >= 3
                   ifTrue: [ self halt:
                  'More primes than I want to compute ?? '].
                     ].
      ].
```

The first process receives the number of primes it is to find from the Primes process as an argument in its method. Depending on the value of numPrimes, one or two primes are stored into the global array PrimeArray by sending it the message at: put:. Access to the array is not restricted since the algorithm assumes that each process accesses non-overlapping elements of the array. Access to the array is animated along the datapath specified for this process, displaying a black line to indicate an active path, a Datum icon animating along the path from the process to the array view diagrams, and and a black array element when the value is stored. After storing its primes the ending index, endx, is set to the last array index accessed + 1. Figure 5.4 shows PR2 after it has retrieved the number 5 from Prime1 and is storing it in the primes array at index 3.

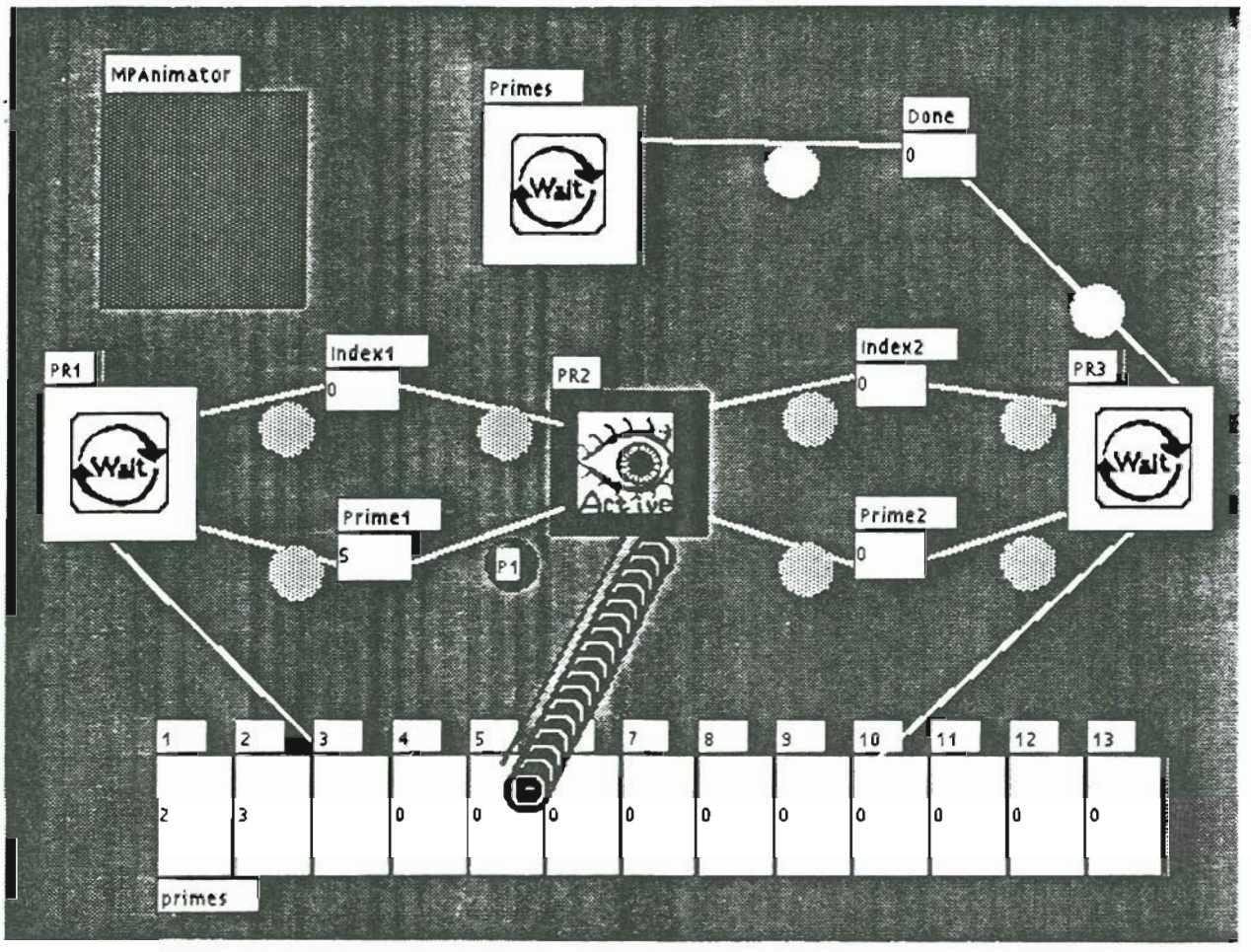

Figure 5.4 PR2 storing prime number 5 into array.

```
        "FOUND MY PRIMES; SEND PRIMEARRAY ENDX ONTO PROCESS2 "
    Index1Sem wait .              " CRITICAL SECTION"
        Index1 set: endx .
    Index1Sem signal .                    " END CRITICAL SECTION"
```

The starting index (Index1) for the next process must be set to endx. Access is restricted by the semaphore Index1Sem. Execution of the messages wait and signal generate a gray and black form for the ISem icon respectively. Similarly, execution of the method set: causes data-path display, animation of the Datum icon, and momentary flashing of the view diagram for Index1.

```
        " LOOP & YIELD FOREVER; GENERATE POSSIBLE PRIMES"
    forever <- true .
    primefull <- true .
    possibleprime <- (PrimeArray at: endx - 1 ) + 2 .
    [ forever ]
      whileTrue: [
        ( self isPrime: possibleprime from: startx to: endx )
          ifTrue: [" SIEVE BY MY NUMPRIMES & IF PRIME SEND ONTO PROCES
              [ primefull ]
                whileTrue: [
                    "CRITICAL SECTION"
                    Prime1Sem wait .
                      ( Prime1 testFor: 0 )
                        ifTrue: [Prime1 set: possibleprime .
                              primefull <- false .
                          Debug transShow:
                         ' PR1 Prime1 = '
                          and:( possibleprime printString ) .
                          ]."if testFor"
                    Prime1Sem signal .
                    " END CRITICAL SECTION"
                    MPAProcessor yield .
                ]. " while primefull"
            ]. "is prime"
```

```
        possibleprime <- possibleprime + 2 .     "generate next prime"
         "reset for next loop and this new possibleprime"
        primefull <- true .
    ]." while forever"
```

The last step for this process is to execute a non-terminating loop in order to generate possible primes and send them on to the next process. Generation of primes is accomplished by adding 2 onto to the last prime generated. For each possible prime generated the method isPrime:from:to: performs the sieving process. This method returns true if the prime sieved properly and false otherwise. New prime numbers must be sent onto the next process by gaining control of the semaphore PrimeSem by issuing a wait message. Once it has the semaphore the contents of Prime1 must be tested to verify that the last prime number stored was retrieved by the next process. If the next process retrieved the previous prime from Prime1 then the new possible prime is stored with the set: possibleprime message. However, if the previous prime was not retrieved then the process must release the semaphore and repeat the testing procedure again after suspending itself. This procedure, of generating new possible prime numbers and sending them onto to the next process by way of the shared message Prime1, is repeated until terminated by the Primes process .

isPrime: aNum from: startx to: endx

```
"checks for prime by dividing by primes
 less than the square root of aNum + 1"
"for primes already stored in classvar
   PrimeArray from startx to from endx"
| sqroot1  i  ok curnum |
sqroot1 <-  ( aNum sqrt ) + 1 .
i <- startx .
ok <- true .
[( i < endx ) & ok ]
     whileTrue: [
       "read only, animation in reverse"
           curnum <- PrimeArray at: i .
           curnum > sqroot1
             ifTrue: [ ok <- false. ]
              ifFalse: [(aNum rem: curnum) == 0
                           ifTrue: [ ^ false |.  ].
           i <- i + 1 .
       ].
     " if arrived here then it's prime "
 ^ true
```

The procedure for determining if a number is prime is performed by this method. It sieves by dividing by primes (stored in the global PrimeArray) less than the square root of the number + 1. It uses prime numbers for the range of array indices that a particular process has access. It executes a loop testing for a remainder of 0; if the result of the division has a remainder of 0 then the number is not prime and the method returns false to the process. If it executes the loop successfully then the number is prime (as far as the current process has sieved) and a value of true is returned.

The second process executes the method primesProcess2: it receives its starting index in PrimeArray; collects primes from Prime1 and stores them in PrimeArray; sends its ending array index onto the next process; and finally collects possible primes from the first process, sieves the number by its own primes, and sends them onto the next process.

```
primesProcess2

  | startx endx possibleprime forever
    prime1full endprime thisprime prime2full |
      "work  for process  PR2 "
      "LOOP & YIELD UNTIL I GET PRIMEARRAY STARTX FROM PROCESS1
  forever <- true .
  [ forever ]
    whileTrue: [
        "  CRITICAL SECTION"
      Index1Sem wait .
        ( Index1 testFor: 0 )
            ifTrue: [Index1Sem signal .
                    MPAProcessor yield . ]
            ifFalse: [startx <-  Index1 get .
                    Index1 set: 0  .
                    forever <- false .
                    Index1Sem signal .
            ]."if testFor"
         " END CRITICAL SECTION"
    ]." while forever"
```

The process suspends (state of waiting) itself each iteration of the loop until it finds a non-zero value in Index1. Access to Index1 is restricted by matching wait and signal messages to the semaphore Index1Sem [Figure 5.5].

"COLLECT POSSIBLE PRIMES FROM PR1,

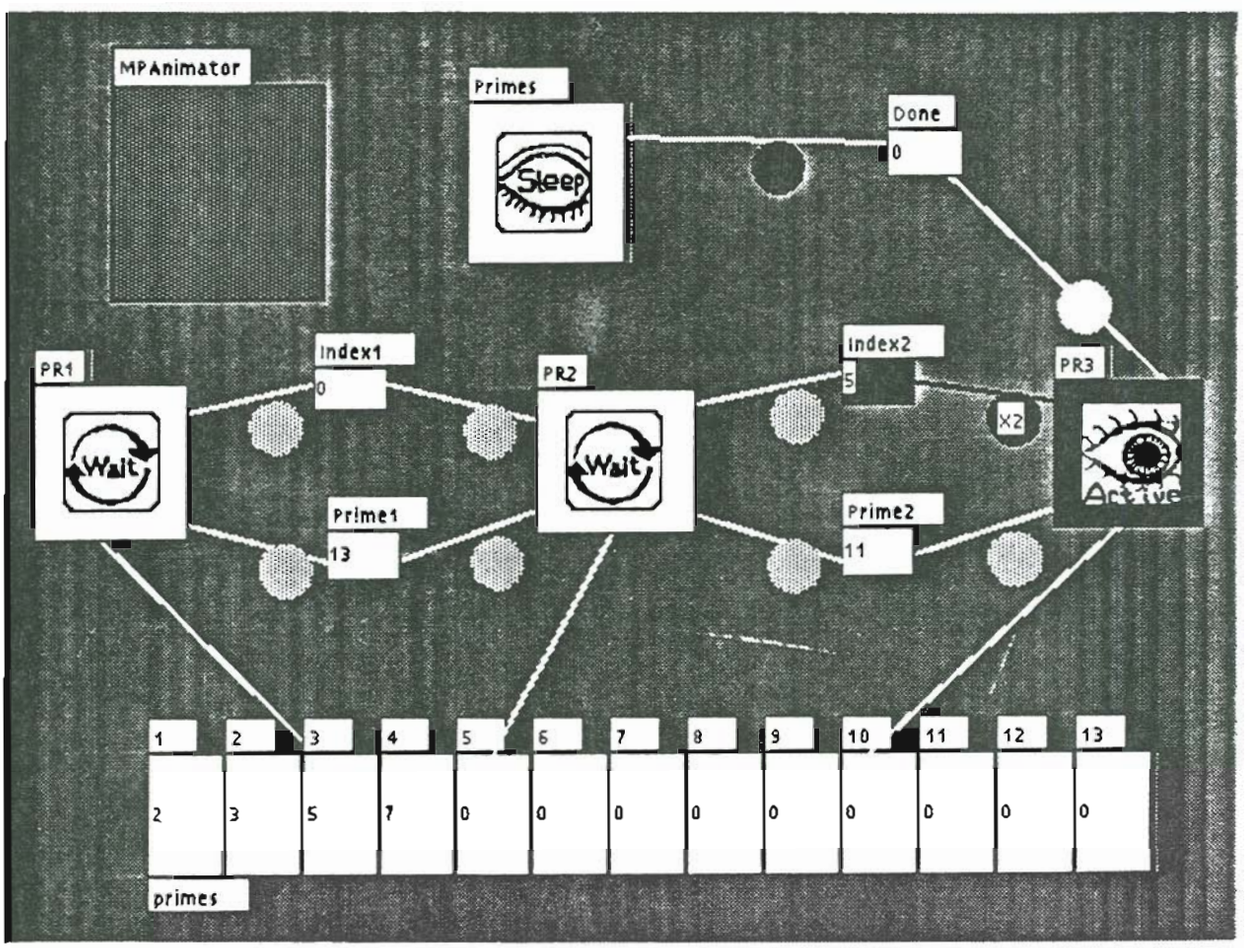

Figure 5.5 PR3 accessing Index2, Primes is suspended on semaphore for Done.

```
    LOOP & YIELD UP TO PR1'S LAST PRIME SQUARED"
endx <- startx .
endprime <- ( PrimeArray at: ( startx-1) ) squared .
forever <- true .
[ forever ]
  whileTrue: [
     "CRITICAL SECTION"
   Prime1Sem wait .
     ( Prime1 testFor: 0 )
          ifFalse: [   thisprime <- Prime1 get .
                ( thisprime < endprime )
                ifTrue: [ PrimeArray at: endx
                                  put: thisprime .
                          Prime1 set: 0 .
                          endx <- endx + 1 .
                        ]
                 ifFalse: [
    " stop collecting primes; pass thisprime onto PR3 if isprime "
                        forever <- false .
                        Prime1 set: 0 . ].
          ]."if testFor"
   Prime1Sem signal .
    " END CRITICAL SECTION"
   MPAProcessor yield .
  ]." while forever"
```

Prime numbers are collected from the first process and stored in PrimeArray; sieving is not necessary since the algorithm uses the fact that numbers it receives less than the square of the previous process's last prime, are guaranteed to be prime. Mutual exclusion of Prime1 is provided by matching signal and wait messages to the semaphore Prime1Sem. The main loop is executed until the new possible prime number in Prime1 is greater than or equal to the previous process's squared prime.

```
        "FOUND MY PRIMES; SEND PRIMEARRAY ENDX ONTO PROCESS3 "
              " CRITICAL SECTION"
      Index2Sem wait .
        Index2 set: endx .
      Index2Sem signal .
              " END CRITICAL SECTION"
```

The ending array index is sent onto the next process after gaining control of the semaphore Index2Sem.

```
        "COLLECT POSSIBLE PRIMES;
        SIEVE BY MY NUMPRIMES & IF PRIME SEND ONTO PROCESS3 "
      forever <- true .
      prime1full <- false .
      [ forever ]
        whileTrue: [
          [ prime1full ]
          "not executed first time through; test thisprime first"
           whileTrue: [
                          "CRITICAL SECTION"
                Prime1Sem wait .
                   ( Prime1 testFor: 0 )
                      ifFalse: [thisprime <- Prime1 get .
                              Prime1 set: 0 .
                              prime1full <- false .
                              Prime1Sem signal .
                               ]
                       ifTrue: [  Prime1Sem signal .
                              MPAProcessor yield .
                       ]."if testFor"
                            " END CRITICAL SECTION"
             ]. " while prime1full"
          ( self isPrime: thisprime from: startx to: endx )
              ifTrue: [ prime2full <- true .
                 [ prime2full ]
                   whileTrue: [
                        "CRITICAL SECTION"
                     Prime2Sem wait .
                       ( Prime2 testFor: 0 )
                       ifTrue: ["empty put in next prime, exit loop"
```

```
                    Prime2 set: thisprime .
                    "now set to get next prime1 from PR1"
                    prime1full <- true .
                    prime2full <- false .
                ]."if testFor"
            Prime2Sem signal .
             " END CRITICAL SECTION"
            MPAProcessor yield .
        ]. " while prime2full"
    ]
  ifFalse: [ prime1full <- true ] . "is prime"
]." while forever"
```

There are two inner loops inside the outer loop which are executed until terminated by the Primes process. The first inner loop (not executed the first time through) simply tests Prime1 for a non-zero value and retrieves it, while the second loop sieves by the current process's primes and passes valid numbers onto the next process in the pipeline. [Figure 5.6] Prime1 contain valid possible prime numbers from the first process and Prime2 contains valid sieved primes that this process is sending onto the third process. Mutual exclusion is maintained to these shared messages by appropriate semaphores.

This is the last process in the pipeline and executes the method primesProcess3. Its actions are similar to the previous process except that after it has received its starting index and collected all valid primes it sends a message to the main Primes process that work is done.

```
primesProcess3
  | startx endx forever endprime thisprime  test |
```

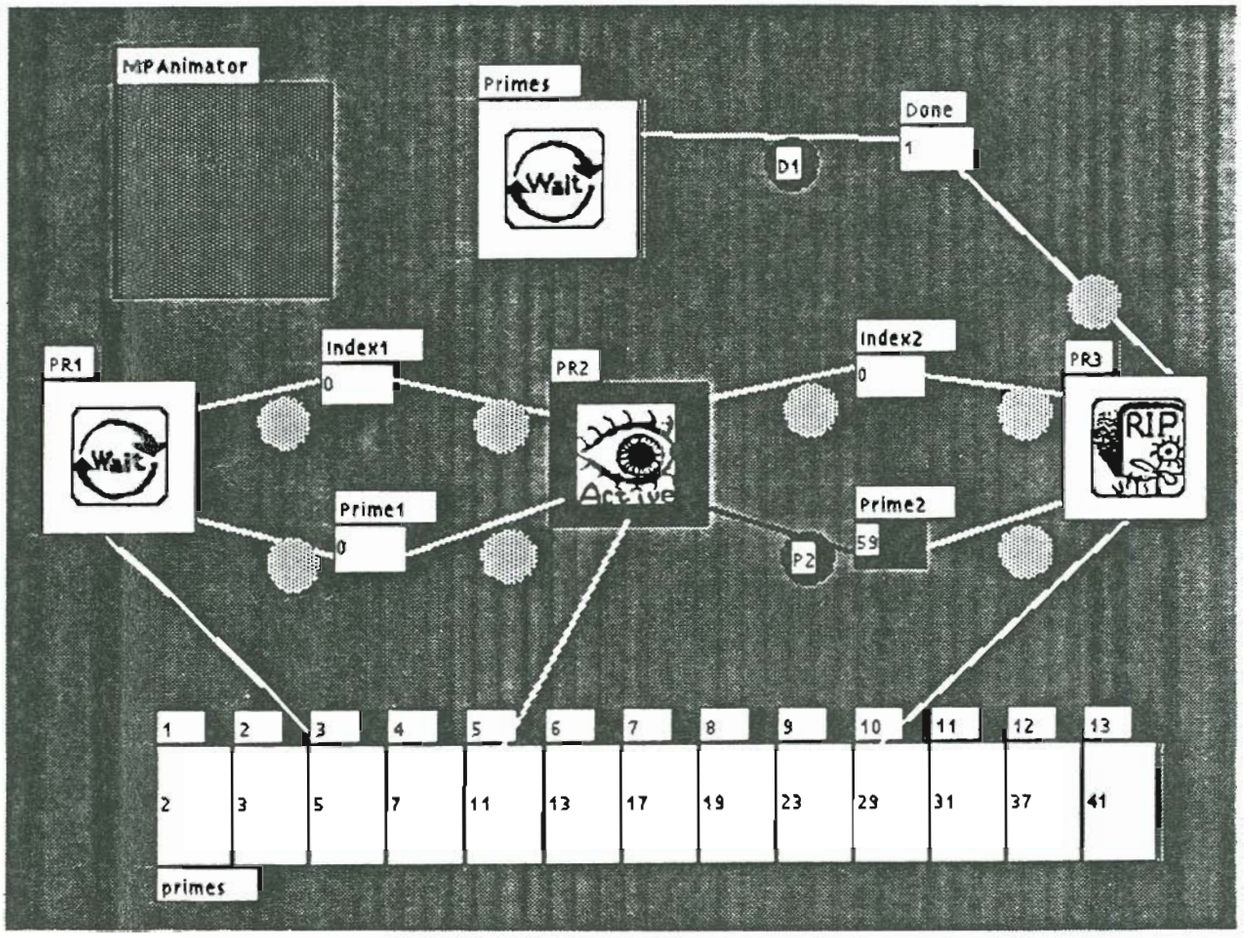

Figure 5.6 PR2 storing 59 into shared data. Primes has semaphore D1 and has been scheduled for execution.

```
"WORK FOR PR3; COLLECTS PRIMES & NOTIFIES MAIN THAT WE'RE DOI
"LOOP & YIELD UNTIL I GET PRIMEARRAY STARTX FROM PROCESS2 "
forever <- true .
[ forever ]
  whileTrue: [
        " CRITICAL SECTION"
    Index2Sem wait .
      ( Index2 testFor: 0 )
        ifTrue: [  Index2Sem signal .
                   MPAProcessor yield . ]
        ifFalse: [ startx <- Index2 get .
                   Index2 set: 0 .
                   forever <- false .
                   Index2Sem signal .
          ]."if testFor"
       " END CRITICAL SECTION"
    |." while forever"
    " COLLECT POSSIBLE PRIMES FROM PR2,
     LOOP & YIELD UP TO PR2'S LAST PRIME SQUARED"
endx <- startx .
endprime <- ( PrimeArray at: ( startx-1) ) squared .
    forever <- true .
    [ forever ]
      whileTrue: [
            "CRITICAL SECTION"
        Prime2Sem wait .
          ( Prime2 testFor: 0 )
            ifFalse: [  thisprime <- Prime2 get .
              ( thisprime < endprime )
                ifTrue: [ PrimeArray at: endx
                                put: thisprime .
                    Prime2 set: 0 .
                    endx <- endx + 1 .
                ]
                ifFalse: [
             "stop collecting primes; Notify Main we're done"
                    forever <- false .
                    Prime2 set: 0 . ].
            ]."if testFor"
         Prime2Sem signal .
         " END CRITICAL SECTION"
         MPAProcessor yield .
```

]." while forever"

The code for collecting primes is identical to that of the previous process except for the semaphores and shared messages referenced. [Figure 5.7]

```
"NOTIFY MAIN THAT WE'RE DONE COMPUTING PRIMES"
"CRITICAL SECTION BUT I SHOULDN'T HAVE TO WAIT, JUST SIGNAL
( Done testFor: 0 )
    ifTrue: [  forever <- false .
               Done set: 1 .
        ]."if testFor"
DoneSem signal .          " END CRITICAL SECTION"
```

The final action for this process is to signal the main Primes process that it has collected all of the primes. Mutual exclusion is not maintained here since Primes has suspended (state of sleep) itself when it executed a wait on the semaphore DoneSem [Figure 5.5]. When DoneSem was declared in the method initSemaphores there was no initial signal message issued, consequently when Primes issued a wait to DoneSem it was suspended until a corresponding signal was issued (as this process did at the end of this method).

5.3. Performance Monitoring

MPA supports rudimentary performance monitoring. If performance is a primary goal of an application, then metrics that may be of interest are:

Total process computation time.

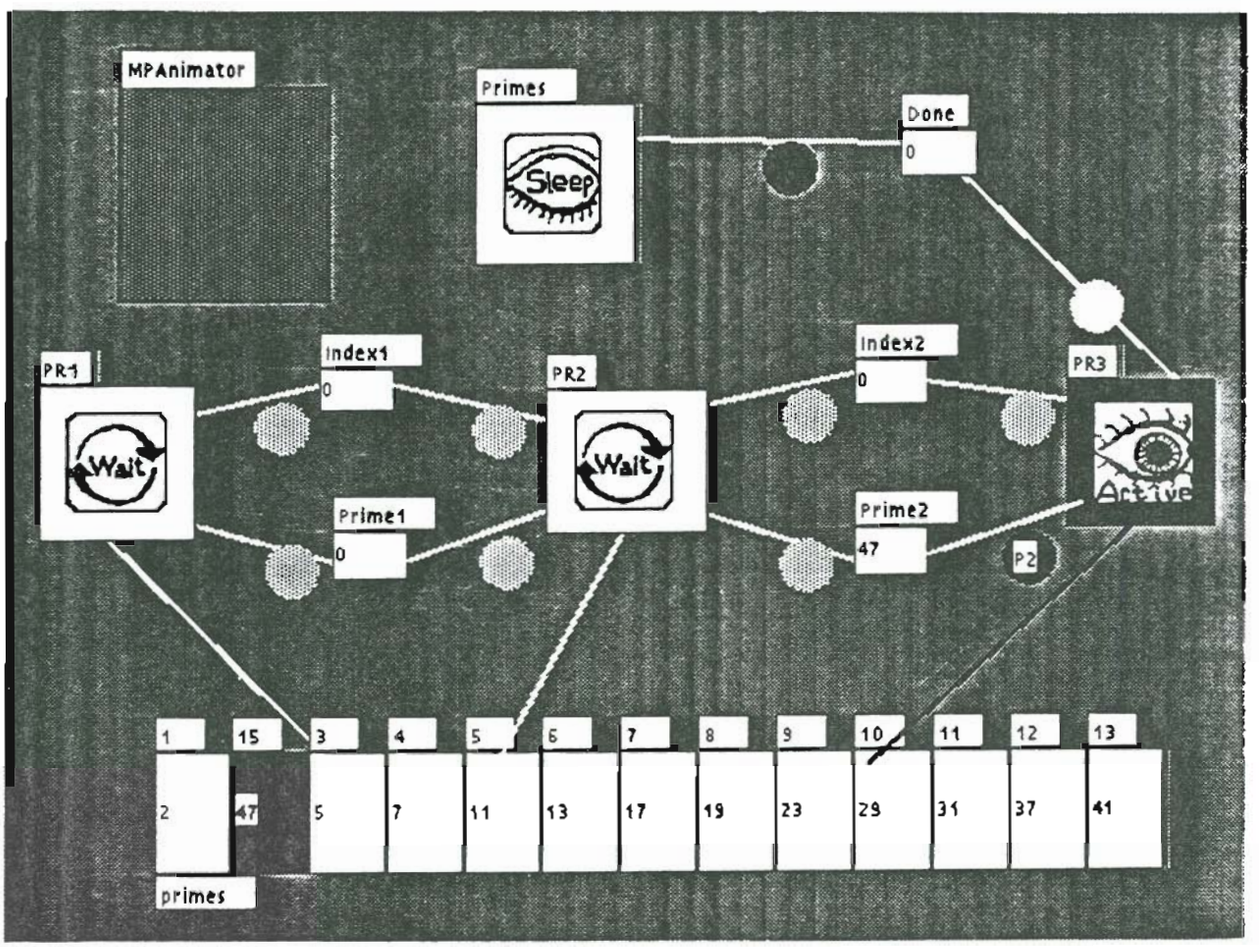

Figure 5.7 PR3 storing prime number 47 into array.

Time spent waiting for the CPU.
Time spent blocked on a semaphore.
Amount of inter-process synchronization.

Normally these would be presented as a statistic or graphical chart to represent the information collected. MPA does not currently provide such facilities, although they would be relatively easy to implement. However, a more interesting and intuitive analysis is provided by means of the *behavioral displays* that are presented during execution of an MPA application. By observing the behavior of the program during execution, a programmer can develop an intuitive feel for performance bottlenecks in an MPA program. This often requires repeated observation of the program's behavior.

For example, after the bugs were removed from the Sieve program, observation of its behavior enabled the author to detect subtle bottlenecks in the pipeline. When the second and third processes are collecting there primes from the previous process, they gain control of the semaphore for the prime shared data and keep it until after they have stored the number in the shared primes array. A more efficient algorithm would zero the prime shared data immediately after access, and then release the semaphore before storing the prime number in the array.

Another bottleneck occurs as the pipeline is filling up and when the program is terminating. At startup, the third process wastes execution cycles checking for its starting array index, which the second process hasn't set. At termination, the first and second processes are wasting CPU cycles, by continuing to fill the pipeline, while the third process is signaling the main process that all work has finished. This is caused by a lack of synchronization in the algorithm. If each process waited on semaphores (suspended) for execution, then they would produce only when the appropriate process signaled. For example, at startup the third process would suspend itself on the array index semaphore until the second process had stored the starting index and sent a signal message to the semaphore. The third process would be scheduled to execute knowing that it has data to consume.

6. BEHAVIORAL ABERRATIONS

The primary design goal for MPA was to provide programmers assistance to:

1) Pinpoint where *bugs* occur in a program.
2) Determine which process or processes are responsible.
3) Provide some indication of the nature of the problem for a particular bug.

Of course, it should do all of the above *graphically*, so that the programmer can observe the bugs as they occur. This assumes that the programmer recognizes what is correct and incorrect for program behavior during the execution of a program. This section presents some snapshots of the MPA environment during execution of the Sieve program. The code responsible for the bugs is also presented. The previous section presented what a correct execution of the program should look like; this section presents an incorrect version that exhibits what I call *behavioral aberrations*. The bugs discussed here are actual bugs that were present in an earlier version of the Sieve program. Unfortunately, observations of an actual MPA animation cannot be adequately presented by static snapshots. Infinite loops were a problem that were handled by loop counters that terminated the loop when the counter value was exceeded (this code is not shown).

6.1. Failure to release a Semaphore

The first indication of a bug in the program was exhibited by PR3 in Figure 6.1. PR2 is the active process and is about to retrieve the number 11 from Prime1 (inserted by PR1). It has control of the semaphore P1 that provides mutual exclusion to the shared data. PR1 and PR3 are waiting there turn to execute. However, PR3 still has control of the semaphore X2 that provides protected access to Index2 (the starting array index for PR2). The algorithm calls for access and release of the semaphores that provide mutual exclusion for the array indices and prime number shared data.

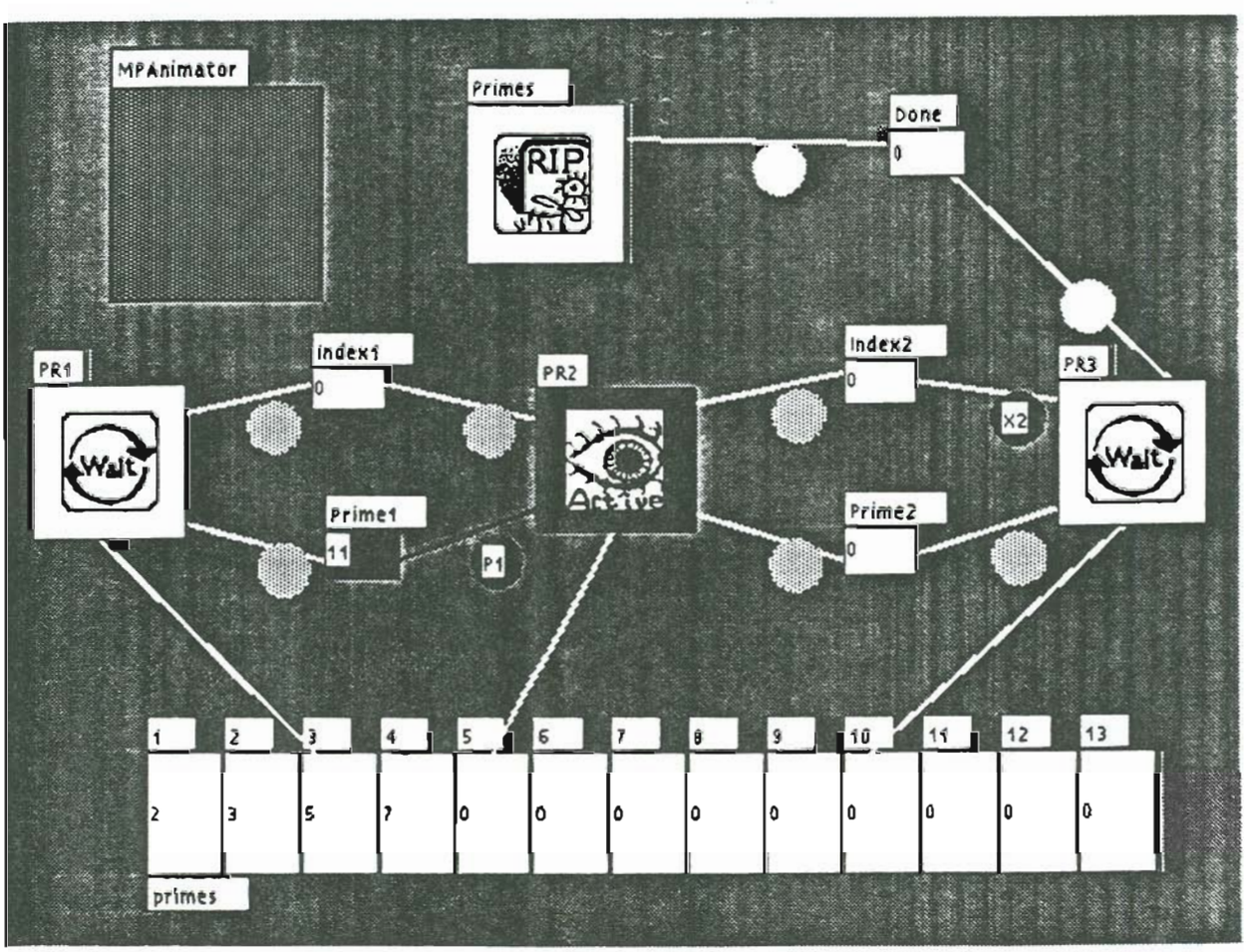

Figure 6.1 Failure to release semaphore X2.

The piece of code in PR3 that caused the problem was:

```
forever <- true .
[ forever ]
  whileTrue: [
        " CRITICAL SECTION"
    Index2Sem wait .
      ( Prime2 testFor: 0 )
        ifTrue: [ MPAProcessor yield . ]
        ifFalse: [ startx <- Index2 get .
                   Index2 set: 0 .
                   forever <- false .
          ]."if testFor"
    Index2Sem signal .
        " END CRITICAL SECTION"
   ]." while forever"
```

This code was intended to loop forever until the contents of Index2 was non-zero. After issuing the wait message to Index2Sem, Prime2 is tested for 0. For the first few iterations of the loop it is 0 and executes a yield (suspending itself), without releasing the semaphore. After collecting the primes 5 and 7 from PR1, PR2 attempts to send the starting array index (5) onto PR3 by issuing a wait message to the semaphore that PR3 is holding. This causes PR2 to be suspended. Figure 6.2 shows the execution of PR2 after it has retrieved the prime number 11 and has issued a wait message to the semaphore X1. Since PR3 still has the semaphore, PR2 is suspended until PR3 releases the semaphore with a signal message. PR1 has generated another possible prime and inserted it into Prime1. All is not lost, since PR3, on its next execution

after the yield issues a signal to Index2Sem. This signal message causes scheduling of PR2. However, PR3 now executes a wait message at the top of the loop and is itself suspended. When PR2 becomes the active process it sets Index2 to 5 for PR3 and then issues a signal, which has the effect of scheduling PR3. Sometimes a program makes progress, in spite of the programmer.

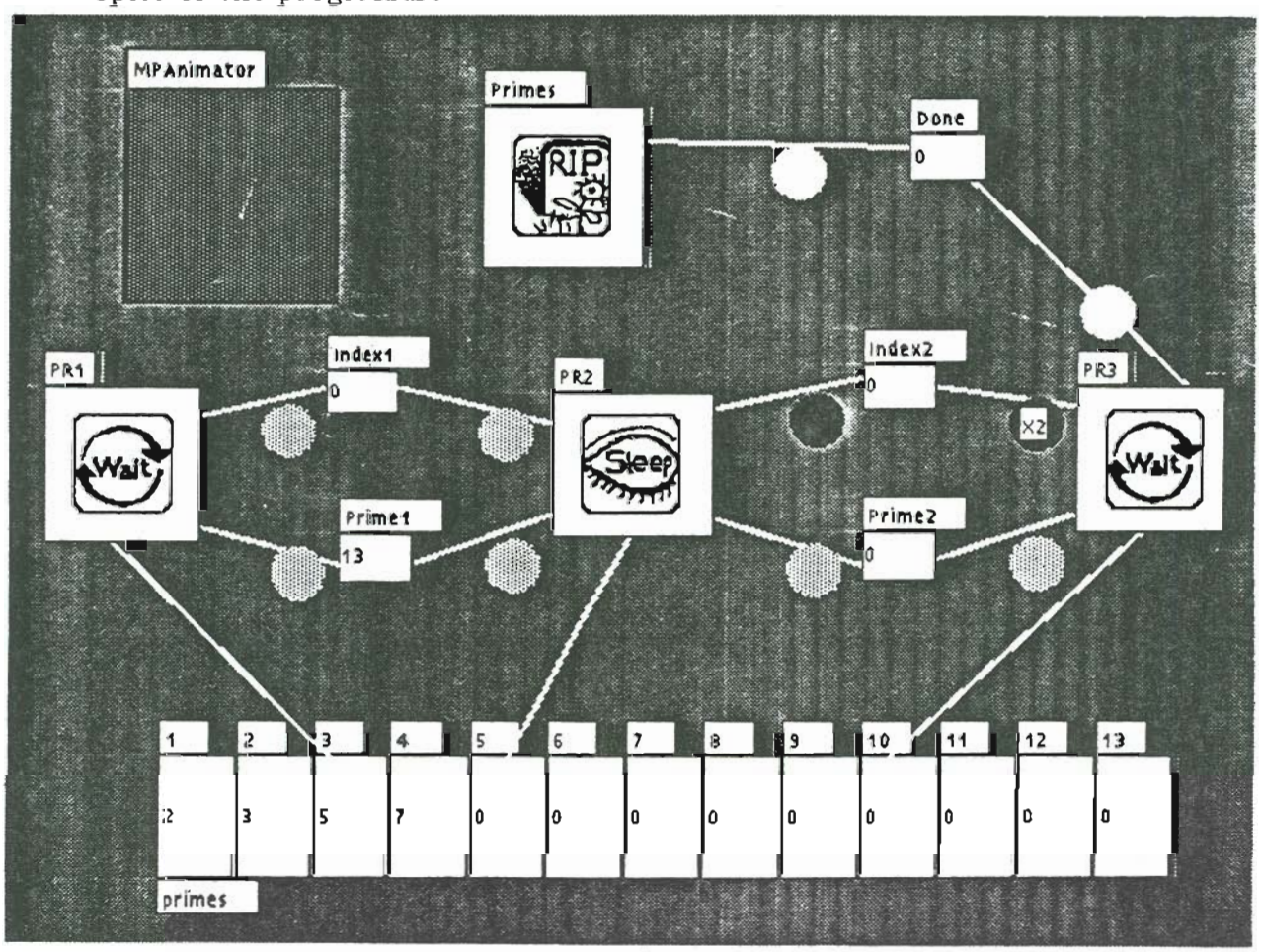

Figure 6.2 PR2 suspended. Primes terminated early.

There is another problem with the program, as shown in Figure 6.2; the Primes process has already terminated. The algorithm calls for

Primes to wakeup after PR3 has signaled and terminate the other processes. The Primes process was missing the code that checked Done for a non-zero value and terminates the other processes. This error was caused by an omission of code.

6.2. Redundant Prime Numbers

PR3 does release control of the semaphore and PR2 does send the number 5 onto PR3 for its starting array index. The next bug that becomes apparent in the observation of the program occurs when PR3 repeatedly stores the number 11 into the primes array. Figures 6.3 and 6.4 viewed together show the developing story; PR1 stops sending new prime numbers onto PR2 because PR2 no longer retrieves new primes from Prime1. PR2 continually stores the prime number 11 into Prime2, which PR3 dutifully retrieves and stores into the primes array.

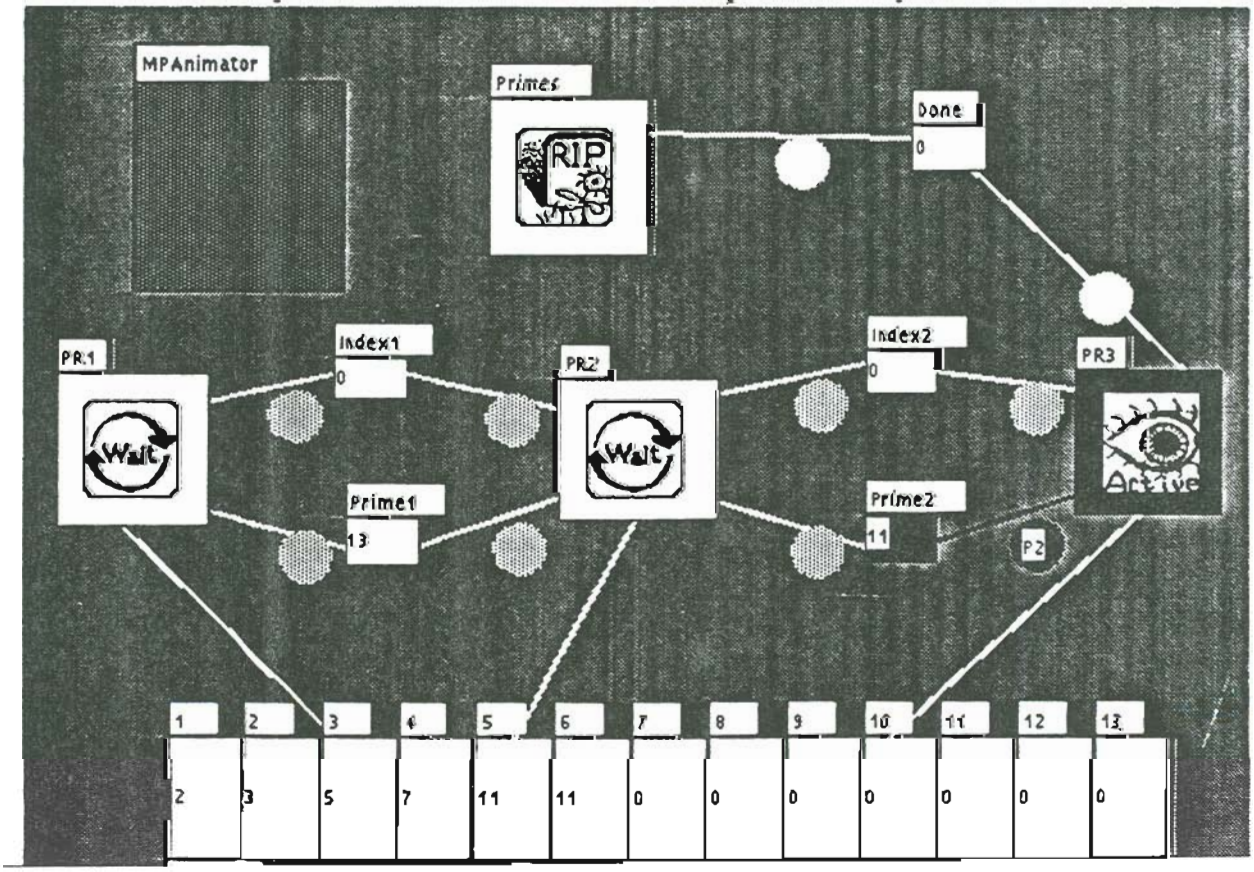

Figure 6.3 PR1 stops sending new prime numbers. PR2 only sends the number 11.

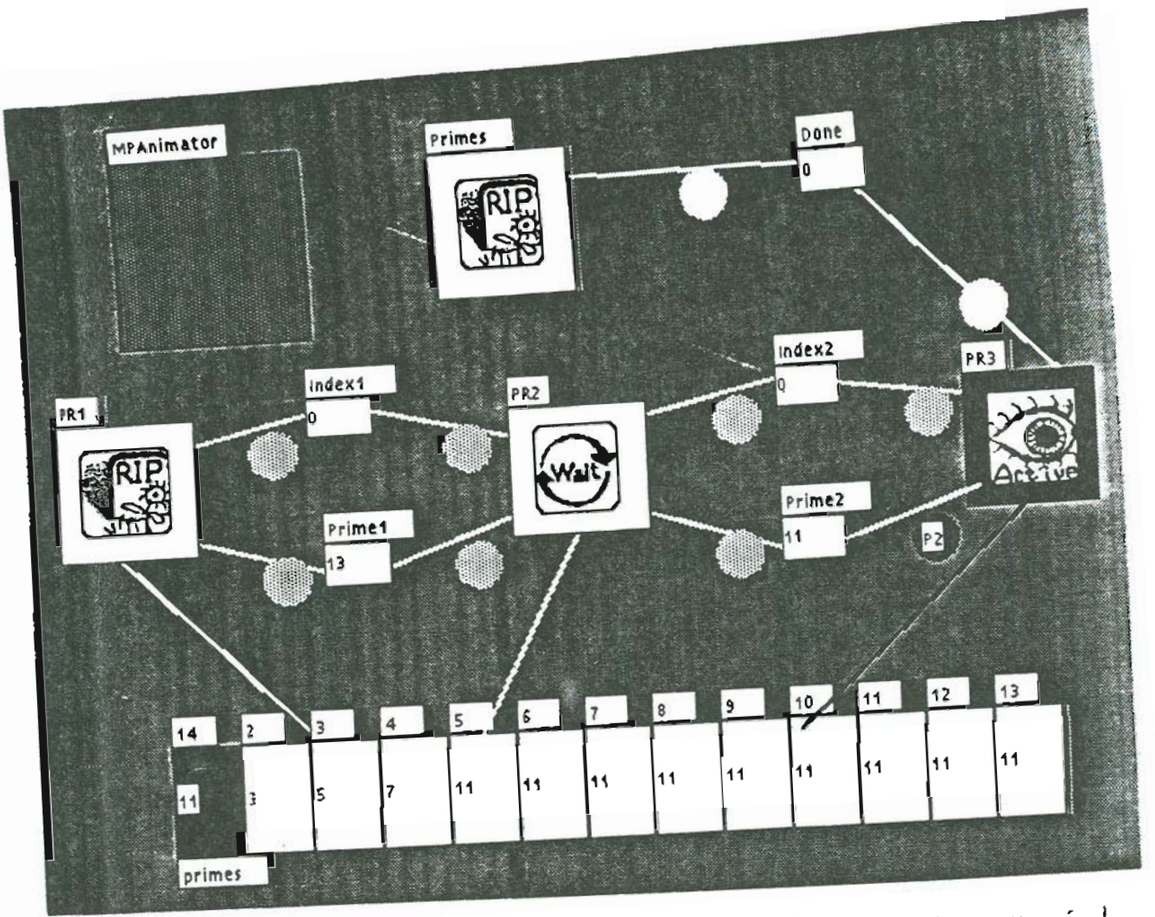

Figure 6.4. PR3 repeatedly stores prime number 11 into array.

We appear to have a problem with generating new prime numbers. PR1 appears to be behaving, since it shouldn't store another prime number into Prime1 if PR2 hasn't retrieved it. PR3 appears ok, since it is retrieving and storing prime numbers. PR2 seems to be the immediate culprit; it isn't retrieving new prime numbers from PR1. The code that was causing the problem in PR2 was as follows:

loops depended on different objects (prime1full and prime2full).

6.3. Correct Mutual Exclusion, Wrong Shared Data

This is an example of a bug, that for this particular algorithm, did not cause any problems. This is an example where slowing down the animation is usefull, because the duration of the behavior is short. The code that causes the problem was contained in PR2 and PR3 as follows:

```
Index1Sem wait .
   ( Prime1 testFor: 0 )
      ifTrue: [ MPAProcessor yield . ]
      ifFalse: [ startx <- Index1 get .
                 Index1 set: 0 .
                 forever <- false .
        ]."if testFor"
Index1Sem signal .
```

The code is the same in PR3 except for names of the semaphore and shared data. The problem is that the correct semaphore is referenced for mutual exclusion, but the wrong shared data (Prime1) is tested (without mutual exclusion). This does not cause an error during execution because the contents of Prime1 is set after Index1 is set. During animation, the testFor: message causes the shared data view to display black. The link is not displayed and the Datum icon is not animated. This was a implementation decision that made it somewhat difficult to observe this type of error and required slowing the animation down.

6.4. Sending The Wrong Message To Another Process

The last major problem that was observed for this version of the program, was caused by PR3 setting the shared data Done to the wrong value [Figure 6.5].

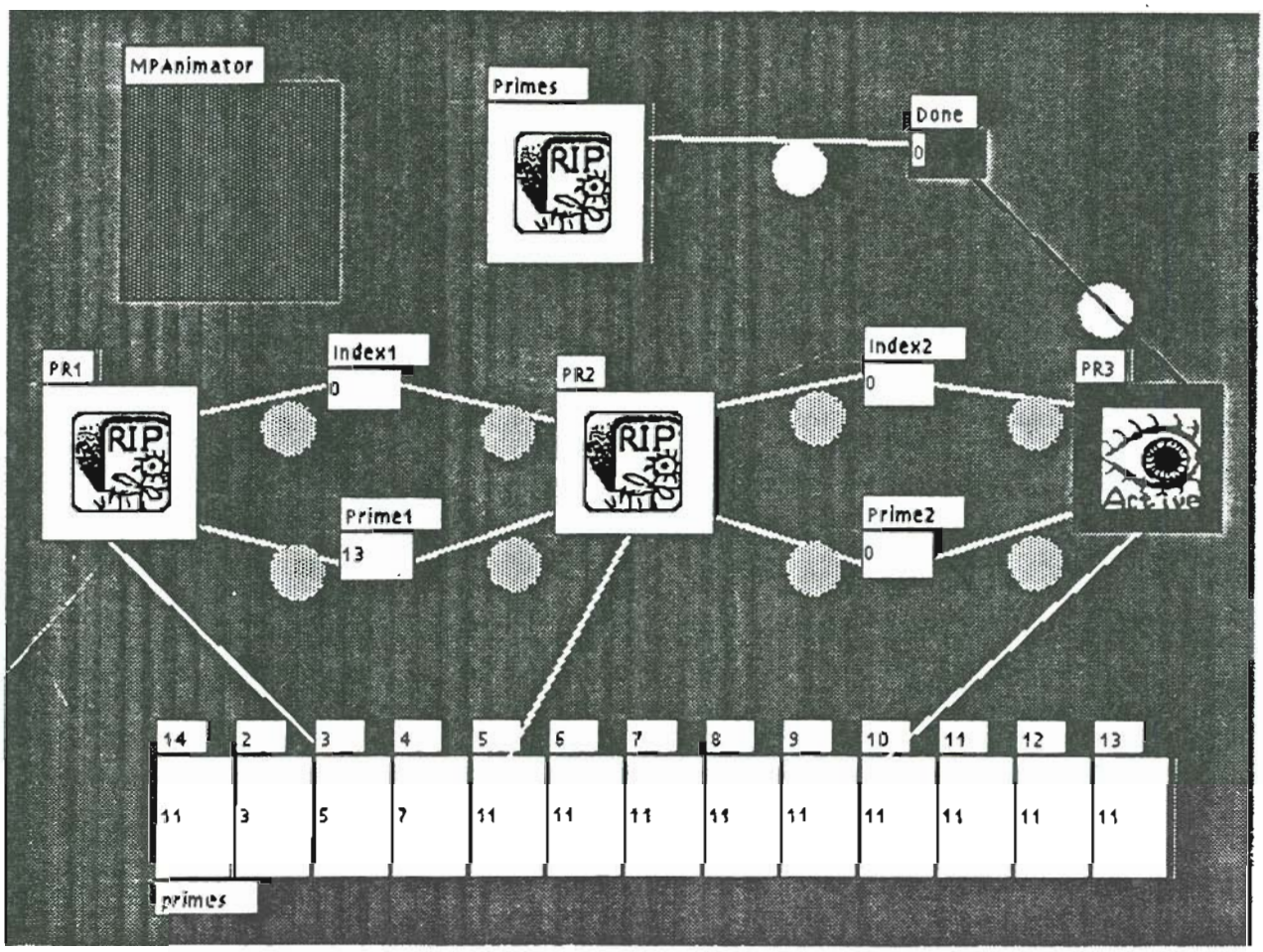

Figure 6.5 PR3 sets Done to incorrect value.

The error was one of sending Done the message set:0 rather than set:1. It didn't matter in this case, since Primes had already terminated.

However, if Primes was testing for a non-zero value as follows:

```
    "WAIT FOR SIGNAL FROM PR3 THAT WE'RE DONE"
forever <- true .
| forever ]
  whileTrue: [
    " CRITICAL SECTION"
    DoneSem wait .
        ( Done testFor: 0 )
          ifTrue: [  DoneSem signal .
                     MPAProcessor yield . ]
           ifFalse: [  Done set: 0  .
                       forever <- false .
                       DoneSem signal .
                   ]."if testFor"
    " END CRITICAL SECTION"
  ]." while forever"
```

In this case Primes would not receive the correct message, and would loop forever as would PR1 and PR2 since Primes is responsible for their termination.

6.5. Miscellaneous Behavioral Bugs

There are a variety of bugs possible in programs such as the Sieve version presented here. Some of the other bugs that can be observed easily within MPA are:

1) Initialization of shared data and semaphores. Shared data that are not initialized display a ? symbol. Semaphores display as white forms.
2) Creation of too few processes. A process that has not been created displays the nil icon |Figure 4.1|.

7. SUMMARY

Myers [Myers 86] states that program visualization systems have the following problems:

1) It is difficult to find correct graphical representations for a data structure.
2) It is hard to program the system to produce the graphics once a representation has been chosen.
3) It is difficult to get large amounts of data on the screen.
4) There is a layout problem: deciding where to place all the graphical representations that may have lines connecting them.
5) In animations, there is the problem of deciding when to update a display. Timing is critical to produce useful animations.

MPA has attempted to deal with these critical problems by providing: a basic animation toolkit, reusable graphical objects, programmer directed layout, readily extensible Smalltalk-based system. There is no easy solution for displaying large amounts of data on standard display screens. This has been mitigated by larger 19 inch bit-mapped screens used on some workstations.

Smalltalk provides for rapid prototyping for environments such as MPA. Smalltalk supports multiple inheritance, data abstraction and encapsulation. There are some problems with Smalltalk as a programming environment. Performance problems related to memory and length of the execution cycle do not support numerical applications well. Smalltalk has not been extensively used commercially and has a steep learning curve. The Model-View-Controller paradigm has caused

problems related to updating display information.

MPA is a prototype, and has satisfied its design goals. However, it has shortcomings that could be improved on:

1) It needs more experimentation to determine if the implementation is appropriate for a wide variety of programs.
2) It does not provide run-time support for more than one program application.
3) The graphical specification interface could be improved to provide for easier specification of an animation and an editor to allow change of an existing animation layout.
4) There is no automated layout support, all layout must be specified by the programmer.
5) There are no run-time statistics provided to monitor performance.
6) There is no provision for recording a script of an execution that could be rewound to various points and replayed.
7) The graphical toolkit for data structures only supports simple variables and arrays.
8) The Smalltalk process scheduler does not provide for pre-emption of processes, which would provide a more realistic simulation for multiprocessing.

Aside from improvements to MPA itself there are other directions for future research. Do simulations such as those described here transfer directly to actual shared memory multiprocessor architectures? In other words, would it be possible to provide a translator that converted Smalltalk and MPA methods of a multiprocessing program into another language (such as Fortran or C) that would execute correctly on some multiprocessing hardware? The simple examples that the author has experimented with in MPA would map easily to a Cray or Sequent architecture.

Hardware support .....

Most operating systems provide for some manner of debugging support during program development. Most of these debuggers are tedious to use and many programmers bypass these tools in favor of their own methods. Programmers would be more inclined to use such tools, if they supported visualization and animation of their programs.

8. REFERENCES

[Bocker, et.al. 86]

Bocker, H., Fischer, G., Nieper,H. The Enhancement of Understanding through Visual Representations. SIGCHI'86 Conference Proceedings, ACM, pp44-50, 1986.

[Bokhari 86]

Bokhari, S. Multiprocessing the Sieve of Eratosthenes. IEEE Computer 20(4) , April 1987.

[Brandis 86]

Brandis, C. IPPM:Interactive Parllel Program Monitor MS thesis, Oregon Graduate Center, 1986.

[Brown and Sedgewick 85]

Brown, M.H., and Sedgewick, R. Techniques for Algorithm Animation. In Proceedings of the 18th Hawaii International Conference on System Sciences, pp104-113. 1985.

[Brown, et al. 85]

Brown, G., Carling R., Herot, C., Kramlich, D., Souza, P. Program Visualization: Graphical Support for Software Development. IEEE Computer 18(8):27-37, August 1985.

[Cox 86]

Cox, B. Object-Oriented Programming An Evolutionary Approach. Addison-Wesley Publ., Reading, MA., 274pp, 1986.

[Duisberg 86]

Duisberg R. Constraint-Based Animation: the Implementation of Temporal Constraints in the Animus System. PhD thesis, University of Washington, 1986.

[Goldberg and Robson 83]

Goldberg, A. and Robson, D. Smalltalk-80: The Language and its Implementation. Addison-Wesley, Reading,Mass. 1983.

[Grafton and Ichikawa 85]

Grafton, R.B. and Ichikawa, T., editors. Visual programming. IEEE Computer 18(8) , August 1985.

[London and Duisberg 85]

London,Ralph L., and Duisberg,Robert A. Animating Programs Using Smalltalk. IEEE Computer 18(8):61-71, August 1985.

[Myers 86]

Myers, B. Visual Programming, Programming by Example, and Program Visualization: A Taxonomy. In Computers and Human Interaction: SIGCHI'86 Conference Proceedings. ACM, 1986.

[OGC 87]

Oregon Graduate Center, Software Engineering Lab: Parallel Processing. Spring Quarter, 1986.

[Reiss 87]

Reiss, S. A Conceptual Programming Environment. Proc 9th Int'l Conf. Software Engineering, 1987.

[Segall and Rudolph 85]

Segall, S., and Rudolph, R. PIE: A Programming and Instrumentation Environment for Parallel Processing. IEEE Software, November, 1985. Proc 9th Int'l Conf. Software Engineering, 1987.

1. APPENDIX A

The following presents the code listing of the methods for the class MPAUserPrimes as described in the text. In Smalltalk: comments are enclosed by double quotation marks ("comment"); <- is the assignment operator; and ^ causes return of a value from a method. References to MPA methods are shown in **boldface** type.

```
MPA subclass: MPAUserPrimes
        instanceVariableNames: ''
        classVariableNames: 'Done DoneSem Index1 Index1Sem
                        Index2 Index2Sem Prime1 Prime1Sem
                        Prime2 Prime2Sem PrimeArray '
        poolDictionaries: ''
        category: 'MPAnimator'

MPAUserPrimes class methodsFor: 'Primes'


    PrimesStartUp

    "adds startup Primes as a newprocess;
       Primes in turn creates other primesprocesses."
      | primes |
           "forkABlock will MPAnimator  initLevel. as needed"
      primes <- MPAProcessor forkABlock: [MPAUserPrimes Primes]
                      newProcessLabel: 'Primes'.
      MPAProcessor resume: primes.
```

```
Primes
  "This is the main Primes process. It invokes other child processes."
  | x y pr1 pr2  pr3 pr4 wans xans
    yans zans pview numprimes test forever |
      " init  all shared data and messages & semaphores "
  self initSharedData .
  self initSemaphores .
  FillInTheBlank request: 'How many Primes should first process create? '
            displayAt: Sensor cursorPoint
            centered: true
            action: [:num | num <- Number readFrom: (ReadStream on: num).]
             initialAnswer: '2' .
      "get number from string"
  numprimes <- num.
  Debug transShow: 'Primes start seed ****' and: numprimes .
  pr1 <- MPAProcessor forkABlock: [self primesProcess1: numprimes ]
              newProcessLabel: 'PR1'.
  pr2 <- MPAProcessor forkABlock: [ self primesProcess2 ]
              newProcessLabel: 'PR2'.
  pr3 <- MPAProcessor forkABlock: [ self primesProcess3 ]
              newProcessLabel: 'PR3'.
      " resume: schedules the processes for exection on cpu "
  MPAProcessor  resume: pr1  .
  MPAProcessor  resume: pr2  .
  MPAProcessor  resume: pr3  .
  "Main yields to other PR1-PR3 processes
        that are waiting for the processor"
  MPAProcessor yield.
    "WAIT FOR SIGNAL FROM PR3 THAT WE'RE DONE"
  forever <- true .
  [ forever ]
    whileTrue: [
      "  CRITICAL SECTION"
       DoneSem wait .
            ( Done testFor: 0 )
              ifTrue: [   DoneSem signal .
                          MPAProcessor yield . ]
               ifFalse: [  Done set: 0  .
                           forever <- false .
                           DoneSem signal .
                        ]."if testFor"
      " END CRITICAL SECTION"
```

```
    ]." while forever"
"TERMINATE OTHER PROCESSES STILL SCHEDULED OR WAITING"
  MPAProcessor terminate: pr1 .
  MPAProcessor terminate: pr2 .
  MPAProcessor terminate: pr3 .
```

```
primesProcess1: numPrimes

    | startx endx possibleprime forever primefull |
  "work  for process  P1 "
  "set first few primes in classvar PrimeArray ; not semaphore restricted"
  startx <- 1 .
  numPrimes == 1
      ifTrue: [  PrimeArray  at: 1 put: 2 .
                 endx <-  2 . ]
      ifFalse: [ numPrimes == 2
                 ifTrue: [  PrimeArray  at: 1 put: 2 .
                            PrimeArray  at: 2 put: 3 .
                            endx <-  3 . ]
                 ifFalse: [ numPrimes >= 3
                            ifTrue: [ self halt:
                            'More primes than I want to compute ?? '|.
                                    ].
               ].
      "FOUND MY PRIMES; SEND PRIMEARRAY ENDX ONTO PROCESS2 "
  Index1Sem wait .                  " CRITICAL SECTION"
      Index1 set: endx .
  Index1Sem signal .                          " END CRITICAL SECTION"
      " LOOP & YIELD FOREVER; GENERATE POSSIBLE PRIMES"
  forever <- true .
  primefull <- true .
  possibleprime <-  (PrimeArray  at:  endx - 1 )  +  2 .
  [ forever ]
    whileTrue: [
      ( self isPrime: possibleprime from: startx to: endx )
        ifTrue: [" SIEVE BY MY NUMPRIMES & IF PRIME SEND ONTO PRO
            [ primefull ]
              whileTrue: [
                  "CRITICAL SECTION"
                  Prime1Sem wait .
                    ( Prime1 testFor: 0 )
                      ifTrue: [Prime1 set: possibleprime .
                               primefull <- false .
                               Debug transShow:
                              ' PR1 Prime1 = '
                               and:( possibleprime printString ) .
                               ]."if testFor"
                  Prime1Sem signal .
```

```
                    " END CRITICAL SECTION"
                    MPAProcessor yield .
                ]. " while primefull"
            ]. "is prime"
        possibleprime <- possibleprime + 2 .     "generate next prime"
         "reset for next loop and this new possibleprime"
        primefull <- true .
]." while forever"
```

```
primesProcess2

  | startx endx possibleprime forever
    prime1full endprime thisprime prime2full |
      "work  for process  PR2 "
      "LOOP & YIELD UNTIL I GET PRIMEARRAY STARTX FROM PROCE
  forever <- true .
  [ forever ]
    whileTrue: [
        "  CRITICAL SECTION"
      Index1Sem wait .
        ( Index1 testFor: 0 )
            ifTrue: |Index1Sem signal .
                    MPAProcessor yield . ]
            ifFalse: |startx <-  Index1 get .
                     Index1 set: 0  .
                     forever <- false .
                     Index1Sem signal .
            ]."if testFor"
         " END CRITICAL SECTION"
    ]." while forever"
      "COLLECT POSSIBLE PRIMES FROM PR1,
      LOOP & YIELD UP TO PR1'S LAST PRIME SQUARED"
  endx <- startx .
  endprime <- ( PrimeArray at: ( startx-1) ) squared .
  forever <- true .
  [ forever ]
    whileTrue: [
        "CRITICAL SECTION"
      Prime1Sem wait .
        ( Prime1 testFor: 0 )
            ifFalse: [  thisprime <- Prime1 get .
                    ( thisprime < endprime )
                    ifTrue: [ PrimeArray at: endx
                                    put: thisprime .
                              Prime1 set: 0 .
                              endx <- endx + 1 .
                            ]
                     ifFalse: [
      " stop collecting primes; pass thisprime onto PR3 if isprime "
                              forever <- false .
                              Prime1 set: 0 . ].
```

```
            ]."if testFor"
        Prime1Sem signal .
         " END CRITICAL SECTION"
        MPAProcessor yield .
      ]." while forever"
        "FOUND MY PRIMES; SEND PRIMEARRAY ENDX ONTO PROCESS3 '
              "  CRITICAL SECTION"
Index2Sem wait .
  Index2 set: endx .
Index2Sem signal .
            " END CRITICAL SECTION"
        "COLLECT POSSIBLE PRIMES;
        SIEVE BY MY NUMPRIMES & IF PRIME SEND ONTO PROCESS3 "
forever <- true .
prime1full <- false .
[ forever ]
  whileTrue: [
    [ prime1full ]
    "not executed first time through; test thisprime first"
     whileTrue: [
                      "CRITICAL SECTION"
          Prime1Sem wait .
             ( Prime1 testFor: 0 )
                ifFalse: [thisprime <- Prime1 get .
                        Prime1 set: 0 .
                        prime1full <- false .
                        Prime1Sem signal .
                         ]
                 ifTrue: [   Prime1Sem signal .
                        MPAProcessor yield .
                 ]."if testFor"
                      " END CRITICAL SECTION"
       ]. " while prime1full"
    ( self isPrime: thisprime from: startx to: endx )
        ifTrue: [ prime2full <- true .
          [ prime2full ]
            whileTrue: [
                 "CRITICAL SECTION"
              Prime2Sem wait .
                ( Prime2 testFor: 0 )
                ifTrue: ["empty put in next prime, exit loop"
                        Prime2 set: thisprime .
```

```
                        "now set to get next prime1 from PR1"
                        prime1full <- true .
                        prime2full <- false .
                    ]."if testFor"
                Prime2Sem signal .
                 " END CRITICAL SECTION"
                MPAProcessor yield .
            ]. " while prime2full"
        ]
    ifFalse: [ prime1full <- true ] . "is prime"
]." while forever"
```

```
primesProcess3
  | startx endx forever endprime thisprime  test |
  "WORK FOR PR3; COLLECTS PRIMES & NOTIFIES MAIN THAT WE'RE
  "LOOP & YIELD UNTIL I GET PRIMEARRAY STARTX FROM PROCESS2
  forever <- true .
  [ forever ]
    whileTrue: [
          " CRITICAL SECTION"
      Index2Sem wait .
        ( Index2 testFor: 0 )
          ifTrue: [  Index2Sem signal .
                     MPAProcessor yield . ]
          ifFalse: [ startx <-  Index2 get .
                     Index2 set: 0  .
                     forever <- false .
                     Index2Sem signal .
            ]."if testFor"
          " END CRITICAL SECTION"
      ]." while forever"
      " COLLECT POSSIBLE PRIMES FROM PR2,
       LOOP & YIELD UP TO PR2'S LAST PRIME SQUARED"
  endx <- startx .
  endprime <- ( PrimeArray at: ( startx-1) ) squared .
      forever <- true .
      [ forever ]
        whileTrue: [
              "CRITICAL SECTION"
          Prime2Sem wait .
            ( Prime2 testFor: 0 )
              ifFalse: [  thisprime <- Prime2 get .
                ( thisprime < endprime )
                  ifTrue: [ PrimeArray at: endx
                                  put: thisprime .
                        Prime2 set: 0 .
                        endx <- endx + 1 .
                  ]
                  ifFalse: [
                "stop collecting primes; Notify Main we're done"
                        forever <- false .
                        Prime2 set: 0 . ].
            ]."if testFor"
          Prime2Sem signal .
```

```
        " END CRITICAL SECTION"
        MPAProcessor yield .
    ]." while forever"
"NOTIFY MAIN THAT WE'RE DONE COMPUTING PRIMES"
"CRITICAL SECTION BUT I SHOULDN'T HAVE TO WAIT, JUST SIGI
( Done testFor: 0 )
    ifTrue: [   forever <- false .
                Done set: 1 .
        ]."if testFor"
DoneSem signal .            " END CRITICAL SECTION"
```

```
initSemaphores

  "Initializes all classvar semaphores used"
  "DoneSem does not get an initial signal ."
  "label sem & signals set to 0"
  DoneSem <- MPASemaphore newSemaphore: 'D1'.
      "label sem & signals set to 0"
  Index1Sem <- MPASemaphore newSemaphore: 'X1'.
  Index1Sem signal . "set initial signal for other processes"
      "label sem & signals set to 0"
  Index2Sem <- MPASemaphore newSemaphore: 'X2'.
  Index2Sem signal .      "set initial signal for other processes"
      "label sem & signals set to 0"
  Prime1Sem <- MPASemaphore newSemaphore: 'P1'.
  Prime1Sem signal .      "set initial signal for other processes"
      "label sem & signals set to 0"
  Prime2Sem <- MPASemaphore newSemaphore: 'P2'.
  Prime2Sem signal .      "set initial signal for other processes"

initSharedData

  "Performs shared data initialization of classvars."
  PrimeArray <- SharedArray newArray: 'primes'
                        size: 100 value: 0.
  Index1 <- SharedData newNumber: 'Index1' value: 0.
  Index2 <- SharedData newNumber: 'Index2' value: 0.
  Prime1 <- SharedData newNumber: 'Prime1' value: 0.
  Prime2 <- SharedData newNumber: 'Prime2' value: 0.
  Done <- SharedData newNumber: 'Done' value: 0.
```

```
isPrime: aNum from: startx to: endx

  "checks for prime by dividing by primes
   less than the square root of aNum + 1"
  "for primes already stored in classvar
     PrimeArray from startx to from endx"
  | sqroot1  i  ok curnum |
  sqroot1 <-  ( aNum sqrt ) + 1 .
  i <- startx .
  ok <- true .
  [( i < endx ) & ok ]
       whileTrue: [
         "read only, animation in reverse"
             curnum <- PrimeArray at: i .
             curnum > sqroot1
               ifTrue: [ ok <- false. ]
                ifFalse: [(aNum rem: curnum) == 0
                              ifTrue: [ ^ false |.  ].
             i <- i + 1 .
          ].
       " if arrived here then it's prime "
   ^ true
```